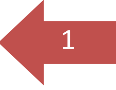
Automi semplici (per davvero)

Rovesti Gabriel



Attenzione

Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.
Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

Introduzione: automi a stati finiti deterministici e concetti base 3
Automi non deterministici/utilizzo di JFLAP/NFA........ 5
Equivalenza di DFA e NFA........ 11
Automi a stati finiti con epsilon-transizioni........ 16
Linguaggi regolari 23
Equivalenza tra FA e RE e conversione per eliminazione di stati 27
Linguaggi non regolari 34
Pumping lemma: esercizi e dimostrazioni........ 40
Grammatiche Context-Free........ 44
Progettare grammatiche context-free 47
Grammatiche context-free/Forme Normali 52
Automi a pila/pushdown automata/PDA........ 57
Da CFG a PDA/Da PDA a CFG........ 62
Linguaggi non context-free........ 73
Macchine di Turing 81
Varianti delle macchine di Turing........ 86
Algoritmi per macchine di Turing 93
Linguaggi decidibili 97
Indecidibilità 102
Non decidibilità e Riducibilità........ 108
Riducibilità 114
Complessità di tempo/Classe P 118
Classe NP 126
NP-Completezza 131
Esercizi NFA/ε-NFA a DFA/NFA a DFA 142
Esercizi Espressioni regolari/ER a NFA/Da NFA ad ER (minimizzazione degli stati) 158
Pumping lemma........ 177
Esercizi Grammatiche context-free 194
Esercizi Forme normali di Chomsky/PDA 205
Esercizi Pumping lemma per linguaggi context-free 211
Esercizi Macchine di Turing e Varianti di Macchine di Turing 244
Esercizi Linguaggi decidibili/Turing-riconoscibili/Linguaggi indecidibili........ 251
Esercizi Linguaggi riducibili tramite funzione ed indecidibili/Riducibilità 273
Esercizi Problemi P/NP 284

Introduzione: automi a stati finiti deterministici e concetti base

La domanda fondamentale di inizio corso è determinare cosa effettivamente possa o non possa fare un calcolatore, un automa o altro, capendo la struttura di un certo problema e quindi risolverlo possibilmente.
Per descrivere un *problema* noi dobbiamo descrivere i possibili input, i possibili output o risultati che coi aspettiamo e la relazione tra di essi.
A tale scopo sappiamo di avere un certo *algoritmo*, quindi una procedura tale a risolvere un dato problema, sulla base di un certo tipo di calcoli e computazioni.
Analogamente, importante valutare correttamente la *complessità* dell'algoritmo, quindi che l'algoritmo risolva effettivamente il problema presente, valutandone poi la *complessità spaziale/temporale.*

Allo stesso modo, poter risolvere un problema riguarda l'astrazione del problema esprimendolo sotto forma di *linguaggio*, visto come *insieme di stringhe*. Le soluzioni trasformano quindi le linee di input con linee di output. L'esempio di problema può essere l'insieme dei numeri primi.
Tutti i processi computazionali possono essere ridotti ad uno tra la determinazione di *appartenenza* ad un insieme, eseguendo una *mappatura* tra insiemi di stringhe.

A questo punto introduciamo il concetto di *automa*, dispositivo matematico astratto in grado di determinare l'appartenenza di una stringa ad un insieme di stringhe e in grado di trasformare una stringa in un'altra stringa. Esso ha tutti gli aspetti di un computer, avendo I/O, una memoria, è in grado di prendere decisioni e può trasformare l'input in output.
Descriviamo in particolare l'aspetto della *memoria*, distinguendo tra *memoria finita ed infinita* e, naturalmente, anche il tipo di accesso alla memoria, distinguendo tra *limitato e illimitato*.

Parliamo ora di *automi a stati finiti*, che sono il modello computazionale più semplice e dispongono di una quantità di memoria *finita*. Gli automi sono usati come *modello* per ricerche di parole chiave, analizzatori lessicali, progettazione di circuiti digitali e scopi similari.
Un esempio semplice è la porta automatica, capendo tramite un sensore se ci sia o meno una persona rilevando la presenza di una persona. In questo caso gli stati sono: *chiusa* o *aperta*.
Quattro possibili input: *fronte/retro/ambo/nessuna.*

In maniera più precisa, possiamo definire alcuni concetti, come il concetto di *alfabeto*, che è un insieme finito e non vuoto di simboli. Il simbolo che lo identifica è il Σ, quello di serie/sommatoria (ad esempio $\Sigma = \{0,1\}$, alfabeto binario, oppure $\Sigma = \{a, b, c, d.... z\}$ insieme delle lettere minuscole, insieme dei caratteri ASCII, ecc.).
Definiamo anche una *stringa (o parola)*, cioè una sequenza finita di simboli da un alfabeto. La *stringa vuota* sarà definita con ε, contenuto in Σ^0. La stringa è composta da un certo numero di simboli e questa è la *lunghezza di una stringa*, definita come $|w|$.

Vi sono poi le *potenze di un alfabeto*, Σ^k = insieme delle stringhe di lunghezza k con simboli da Σ.
Ad esempio: $\Sigma = \{0, 1\}$
$\Sigma^0 = \{\varepsilon\}$ $\Sigma^1 = \{0, 1\}$ $\Sigma^2 = \{00, 01, 10, 11\}$
L'insieme di tutte le stringhe su Σ è denotato da Σ^* (unione di tutte le stringhe matematicamente parlando) e più in generale, dato un alfabeto, un *linguaggio* è ogni sottoinsieme $L \subseteq \Sigma^*$.
Il *linguaggio vuoto* non contiene nessuna parola ed è definito dal simbolo $\emptyset$; si ricordi che il linguaggio vuoto è diverso da una parola vuota.

Un *Automa a Stati Finiti Deterministico (DFA)* è una quintupla $A = (Q, \Sigma, \delta, q_0, F)$, dove:

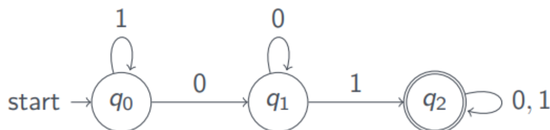
- Q è un insieme finito di *stati*
- Σ è un *alfabeto finito* (= simboli in input)
- δ è una *funzione di transizione* $(q, a) \rightarrow q'$
- $q_0 \in Q$ è lo *stato iniziale*

- $F \subseteq Q$ è un insieme di *stati finali*

In un automa deterministico ogni relazione di transizione ha uno stato di destinazione per ogni simbolo (evento). Non possono esserci relazioni di transizione nello stesso stato che usano lo stesso simbolo.
Una *funzione di transizione* è una funzione con dominio finito, prendendo come finiti l'alfabeto di input e l'insieme degli stati che l'automa può assumere viene rappresentato in forma tabellare come *tabella di transizione* oppure sotto forma di grafo orientato, i cui archi rappresentano le transizioni tra uno stato e l'altro; qui si parla di *diagramma di transizione*.
Possiamo quindi rappresentare gli automi sia come diagramma di transizione che come tabella di transizione.
Esempio pratico:

Esempio: costruiamo un automa A che accetta il linguaggio delle stringhe con 01 come sottostringa

- L'automa come diagramma di transizione:



- L'automa come tabella di transizione:

	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$*q_2$	q_2	q_2

Data una parola $w = w_1w_2 \ldots w_n$, la computazione dell'automa *A* con input *w* è una sequenza di stati $r_0, r_1 \ldots r_n$ che rispetta due condizioni:

1) $r_0 = q_0$ (inizia dallo stato iniziale)
2) $\delta(r_i, w_{i+1}) = r_{i+1}$ per ogni $i = 0, \ldots, n-1$ (rispetta la funzione di transizione)

Diciamo che la computazione accetta la parola w se: $r_n \in F$ (la computazione termina in uno stato finale).
Un DFA *A* accetta la parola *w* se la computazione accetta *w*. Formalmente, il linguaggio accettato da A è $L(A) = \{w \in \Sigma^* \mid A \text{ accetta } w\}$.

<u>*Domande Wooclap*</u>

- Quali sono gli input della funzione di transizione di un DFA? → Uno stato e un simbolo
- Un DFA deve avere un solo stato finale? → Falso
- Qual è l'input del problema "È un numero primo?" → {0,1,2,3,4,5...}
- Quale insieme di stringhe rappresenta il problema nel caso "È un numero primo?" → {2,3,5,7,11,13…}
- Per ogni elemento, stabilire se è carattere, parola, linguaggio:

1) a → carattere
2) abracadabra → parola
3) {abracadabra, apostrofo, a} → linguaggio di tre parole
4) {abracadabra} → linguaggio composto da una parola

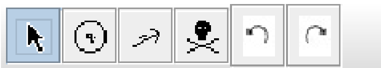
- Dato l'alfabeto $\Sigma = \{0,1\}$ quante sono le stringhe appartenenti al linguaggio? → 16
- Quale dei seguenti linguaggi sull'alfabeto {0,1} contiene un numero infinito di stringhe? → Tutte le stringhe che iniziano con 1
- Dato l'alfabeto $\Sigma = \{0,1\}$ quante sono le stringhe contenute in Σ^0? → ε
- Quante stringhe ci sono nel linguaggio sull'alfabeto {0,1} di tutte le stringhe di lunghezza *n*? → 2^n

- Un DFA possiede una quantità di memoria molto limitata → Vero
- L'insieme di tutti gli stati di un automa si indica con → Q
- Quali sono gli input della funzione di transizione di un DFA? → Uno stato e un simbolo
- Un DFA deve avere un solo stato finale → Falso

Automi non deterministici/utilizzo di JFLAP/NFA

Una volta costruita tutta la computazione è nello stato finale e la parola è *corretta*, essa viene *accettata*, altrimenti se non è uno stato finale l'automa *rifiuta* questa parola.
I linguaggi accettati da automi a stati finiti sono detti *linguaggi regolari.*

Seguono esempi di automi DFA costruibili con il software *JFLAP* (nota: nel menù iniziale, quindi quello con la lista Finite Automaton e seguenti, è possibile impostare il carattere stringa vuota come epsilon; basta cliccare su *Preferences* e poi su *Set the Empty String Character*. Di default è impostato come lambda).



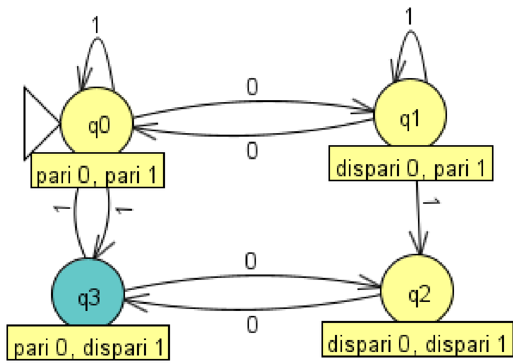
A tale scopo *piccolo tutorial sul software* disponibile in formato jar (descrizione icone da sx verso dx)

1) icona della freccia, che permette di spostarsi liberamente. Cliccando in questo stato con il tasto dx su uno stato è possibile impostarlo come *Initial* o *Final* o eventualmente cambiarne l'etichetta. In questa modalità, inoltre, facendo doppio clic sul valore di una freccia, esso può essere modificato graficamente (viene evidenziato in rosso) e si può immettere un valore senza doverlo eliminare.
2) icona degli stati, messi ad ogni clic del mouse.
3) icona di vettore. Cliccando sull'icona freccia e subito dopo su uno stato senza trascinare la freccia su altri stati, la freccia verrà rivolta verso lo stato attuale (quindi verso lo stato stesso). Altrimenti se si trascina verso un altro stato si imposta la freccia verso quello. Una volta inserito il valore, la freccia viene disegnata. È quindi possibile disegnare sia frecce che vanno avanti ma anche frecce che vanno indietro da uno stato ad un altro in maniera facile.
4) icona del teschio, che elimina frecce e/o stati. Passando la freccia, una volta selezionata questa icona su un valore, nel caso in cui uno stato abbia valori multipli, ne viene eliminato uno singolo, in qualunque posizione esso si trovi.
5) torna indietro
6) torna avanti

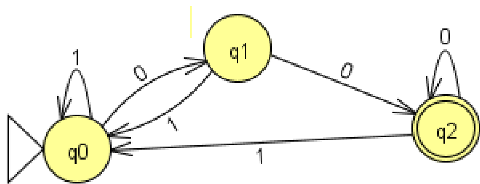
Attenzione anche ad *Automation Size* che funziona da zoom e si può selezionare tutto l'automa con il tasto sinistro e spostarlo poi per l'area di lavoro con il tasto destro.

Sulla barra del menù, cliccando *Input* e poi su *Step by state* è possibile, dando un certo input come stringa, verificare il comportamento effettivo dell'automa. Ciò avviene se è stato impostato almeno uno stato iniziale. Si entra quindi nello stato Simulate e si avanza step by step.
Se il comportamento finale del testo è verde significa che l'automa impostato è corretto; se è rosso, ci sta qualche problema o l'automa è non deterministico (si veda sotto). Per uscire dalla scheda di simulazione attuale e tornare nell'editor si preme la freccia X in alto a dx.
In conclusione, si può salvare l'automa nel formato compatibile con JFLAP oppure in immagine JPG o altro (cliccando su *Convert* per convertirlo, naturalmente, e su File per salvarlo).

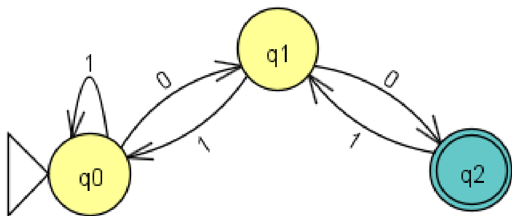
- Insieme di tutte e sole le stringhe con un numero pari di zeri e un numero pari di uni



- *Insieme di tutte le stringhe che finiscono con 00*

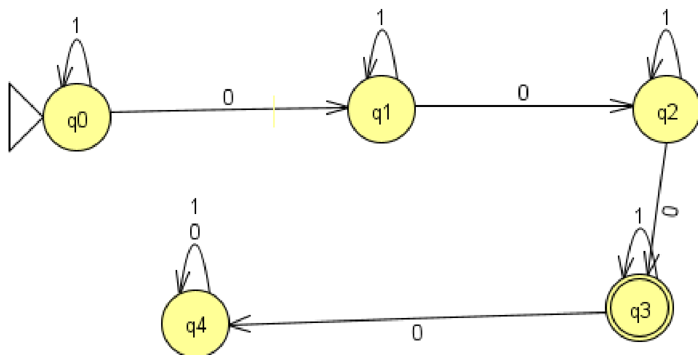


Un'idea non corretta su questo esempio sarebbe:



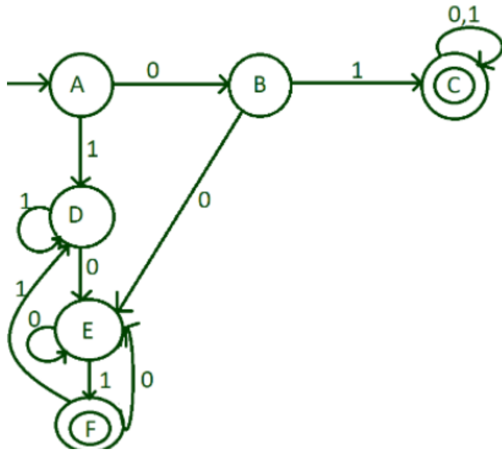
Di fatto è sbagliata come idea perché non c'è una fine vera e propria agli input esistenti (caso rosso su JFLAP come spiegato sopra; l'automa non accetta la stringa fornita); si nota che i due zeri ci sono, ma nel caso avessi un 1 e poi uno 0, ecco che si potrebbe generare un loop per cui la situazione non si sblocca mai.

- *Insieme di tutte le stringhe che contengono esattamente tre zeri (anche non consecutivi)*

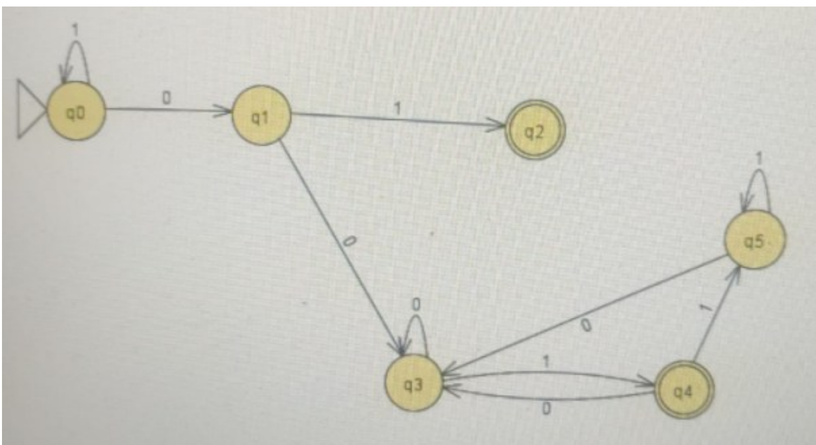


- *Insieme delle stringhe che cominciano o finiscono (o entrambe le cose) con 01 (assegnato dal prof)*

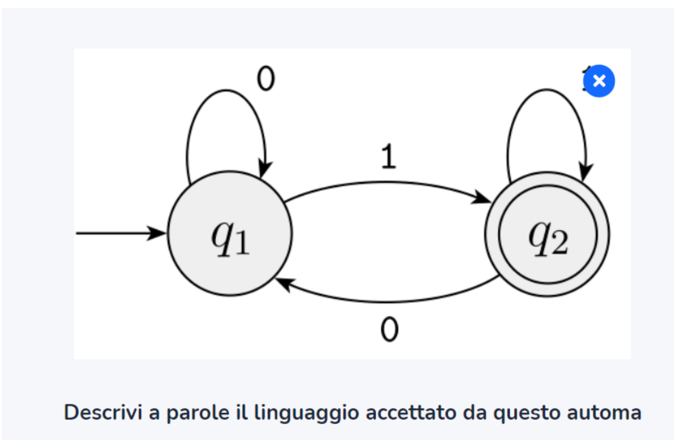
Automi semplici (per davvero)



Altra soluzione:



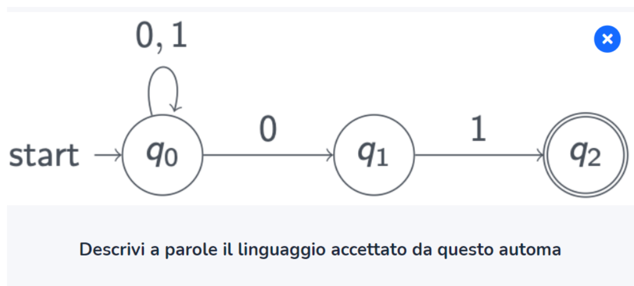
- Quale linguaggio accetta il seguente automa? (c'è un numero nascosto dalla X di chiusura ed è un 1)



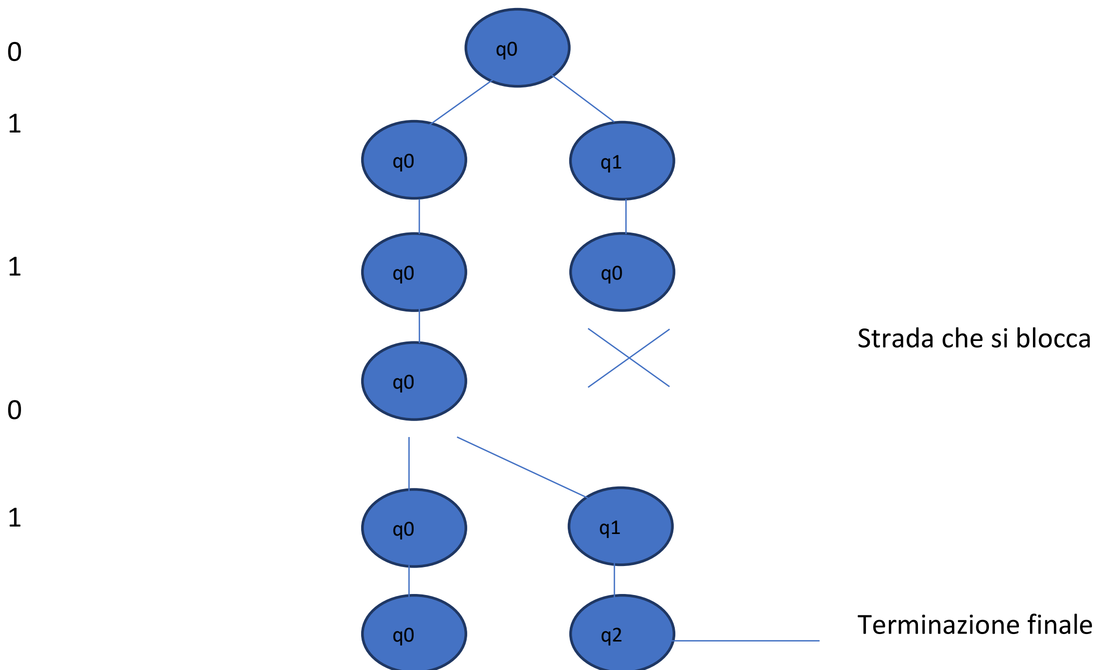
Risposta: Tutte le stringhe composte da 0 e 1 che terminano con 1

Automi semplici (per davvero)

Quale linguaggio accetta il seguente automa?



Risposta (estesa, soluzione più in basso nella pagina):
È un automa non deterministico, perché *la relazione di transizione può avere uno o più stati di destinazione con lo stesso simbolo (evento).*
Se avessi come input 01101, un ramo avrebbe un ramo finale corrispondente, mentre l'altro no.
Gli automi non deterministici o NFA possono essere rappresentati come albero, come si vede qui sotto:



Un *Automa a Stati Finiti Non Deterministico (NFA)* è una quintupla $A = (Q, \Sigma, \delta, q_0, F)$, dove:

- Q è un insieme finito di *stati*
- Σ è un *alfabeto finito* (= simboli in input)
- δ è una *funzione di transizione* che prende in input (q, a) e restituisce *un sottoinsieme di Q*
- $q_0 \in Q$ è lo *stato iniziale*
- $F \subseteq Q$ è un insieme di *stati finali*

Una parola viene accettata se c'è un ramo che arriva ad uno stato finale, altrimenti viene rifiutata; questo avviene se arriva in uno stato che non è finale.
Conclusione: accetta parole che terminano esattamente con 01.

- È un esempio di automa a stati finiti non deterministico:
  - può trovarsi contemporaneamente in più stati diversi
  - le transizioni non sono necessariamente complete:
    - da q_1 si esce solo leggendo 1
    - q_2 non ha transizioni uscenti

  in questi casi il percorso si blocca, ma può proseguire lungo gli altri percorsi

L'NFA che riconosce le parole che terminano con 01 è

$$A = (Q, \{0,1\}, \delta, q_0, \{q_2\})$$

dove δ è la funzione di transizione

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Data una parola $w = w_1w_2 \ldots w_n$, una computazione di un NFA *A* con input *w* è una sequenza di stati $r_0, r_1 \ldots r_n$ che rispetta due condizioni:

1) $r_0 = q_0$ (inizia dallo stato iniziale)
2) $\delta(r_i, w_{i+1}) = r_{i+1}$ per ogni i = 0, . . . , n − 1 (rispetta la funzione di transizione)

Diciamo che una computazione accetta la parola w se: $r_n \in F$ (la computazione *termina in uno stato finale*) A causa del nondeterminismo, ci può essere più di una computazione per ogni parola.
Un NFA *accetta* la parola se esiste una computazione che accetta w (quindi arrivo alla fine correttamente); un NFA *rifiuta* la parola se tutte le computazioni la rifiutano. Le computazioni sono rappresentabili come albero, sulla base dell'esempio di prima, definendo una particolare parola.
A tale scopo, vediamo in dettaglio esempi fatti dal prof e presenti sulle slide come esercizio.

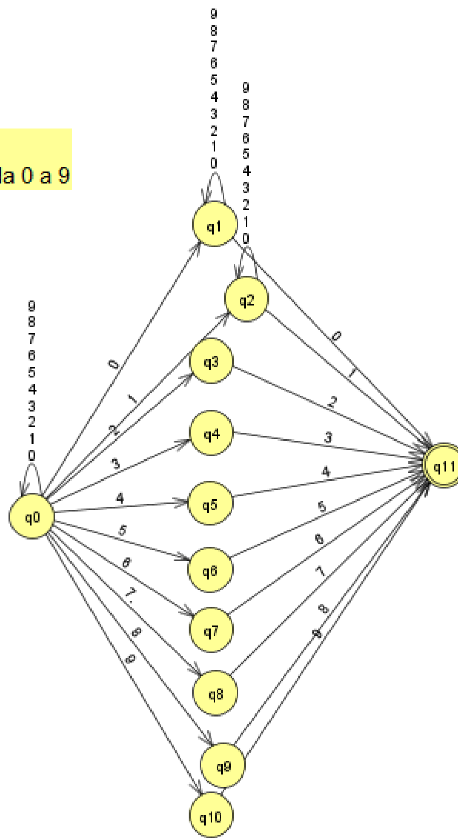
Differenza riassuntiva tra NFA e DFA:

Deterministic Finite Automata	Non-Deterministic Finite Automata
Each transition leads to exactly one state called as deterministic	A transition leads to a subset of states i.e. some transitions can be non-deterministic.
Accepts input if the last state is in Final	Accepts input if one of the last states is in Final.
Backtracking is allowed in DFA.	Backtracking is not always possible.
Requires more space.	Requires less space.
Empty string transitions are not seen in DFA.	Permits empty string transition.
For a given state, on a given input we reach a deterministic and unique state.	For a given state, on a given input we reach more than one state.
DFA is a subset of NFA.	Need to convert NFA to DFA in the design of a compiler.
$\delta : Q \times \Sigma \rightarrow Q$ For example − δ(q0,a)={q1}	**$\delta : Q \times \Sigma \rightarrow 2^Q$** For example − δ(q0,a)={q1,q2}
DFA is more difficult to construct.	NFA is easier to construct.
DFA is understood as one machine.	NFA is understood as multiple small machines computing at the same time.

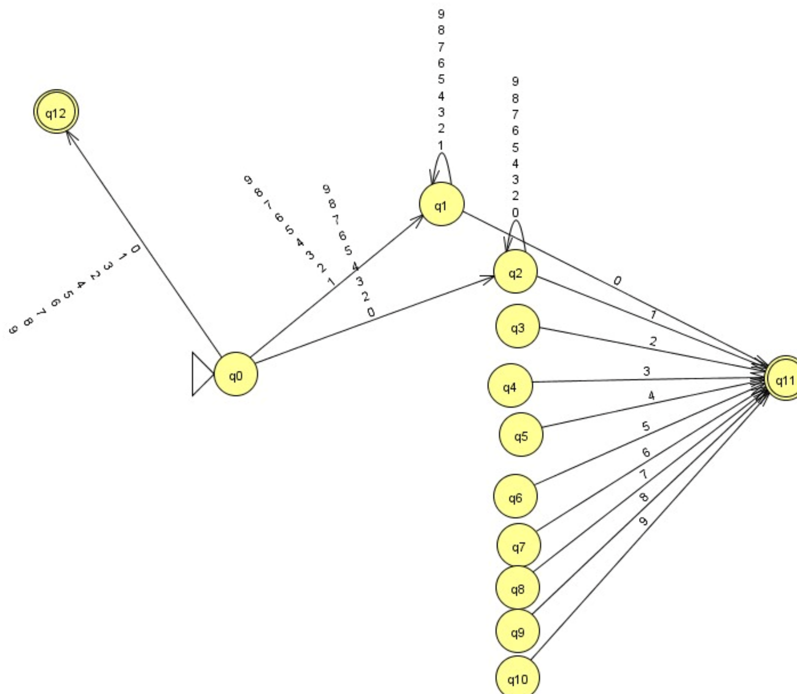
Automi semplici (per davvero)

Primo esercizio:
(L’insieme delle parole sull’alfabeto {0, 1, . . . , 9} tali che la cifra finale sia comparsa in precedenza)

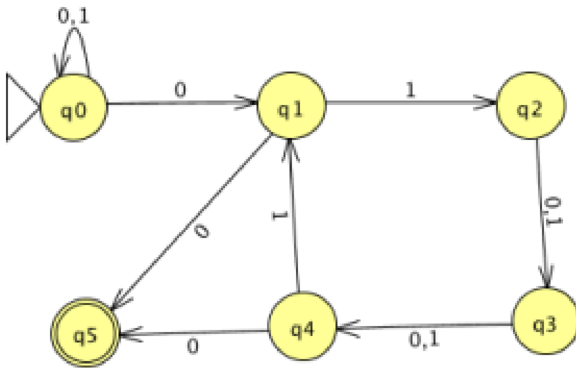
Nota: per essere completi, tutti da q3
in poi dovrebbero avere tutti i numeri da 0 a 9



Secondo esercizio:
(L’insieme delle parole sull’alfabeto {0, 1, . . . , 9} tali che la cifra finale non sia comparsa in precedenza)



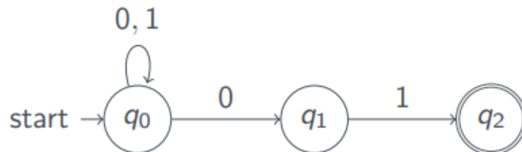
Terzo esercizio:
(L'insieme delle parole di 0 e 1 tali che esistono due 0 separati da un numero di posizioni multiplo di 4 (si considera che 0 sia multiplo di 4))



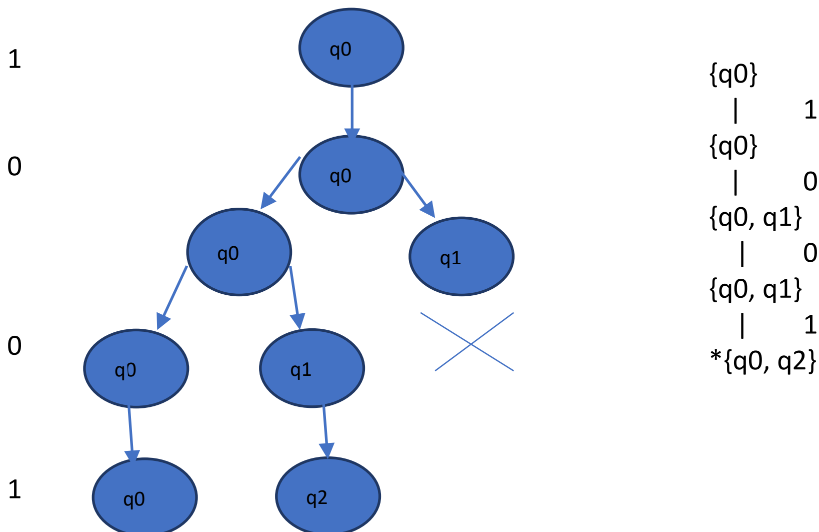
Equivalenza di DFA e NFA

NFA e DFA sono in grado di riconoscere gli stessi linguaggi in quanto, per ogni NFA *N* c'è un DFA *D* tale che L(D) = L(N) e viceversa. Un esempio schematico:

Esempio:



Rappresentato ad albero:



La costruzione è definita equivalentemente come *costruzione a sottoinsiemi*.
Dunque, dato un NFA $N = (Q_N, \Sigma, q_0, \delta_N, F_N)$ costruiremo un DFA $D = (Q_D, \Sigma, S_0, \delta_D, F_D)$ tale che L(D) = L(N)

- $Q_D = \{S : S \subseteq Q_N\}$
  Ogni stato del DFA corrisponde ad un insieme di stati dell'NFA
- $S_0 = \{q_0\}$
  Lo stato iniziale del DFA è l'insieme che contiene solo q_0
- $F_D = \{S \subseteq Q_N : S \cap F_N \neq \emptyset\}$
  Uno stato del DFA è finale se c'è almeno uno stato finale corrispondente nell'NFA
- Per ogni $S \subseteq Q_N$ e per ogni $a \in \Sigma$

$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$$

La funzione di transizione "percorre tutte le possibili strade"

Prende quindi in input un simbolo e ritorna come output uno stato del DFA, costruendo una funzione di transizione che dimostri che posso percorrere tutte le possibili strade.
Matematicamente, si rappresenta l'unione dei singoli stati scorrendo tutti gli stati che stanno in S e mettendo almeno uno di questi come stato destinazione.

Costruiamo δ_D per l'NFA qui sopra:

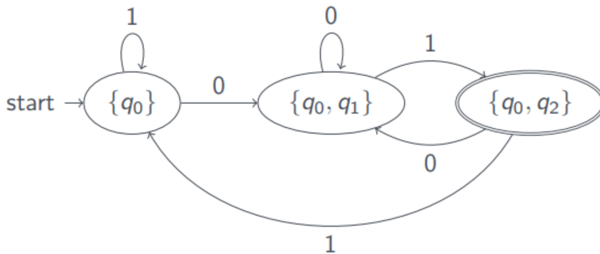
Sotto riportato gradualmente come costruire la tabella corretta di transizione.
Si segnala che nella quinta riga ci sarebbe l'unione di q0 e q1 con l'insieme vuoto, che viene trascurato.
Nelle righe dove ci sono due stati si considerano gli stato da entrambe le parti per 0 e 1, piccola sintesi di come ragionarla.
Note utili:

- *L'asterisco indica gli stati finali e negli stati possibili rientra anche lo stato vuoto. Questo perché, al contrario dei DFA, dove deve sempre esserci una transizione, non è obbligatorio negli NFA e per questo potrebbe non esistere una transizione verso altri stati.*
- *Inoltre, posso intuire i risultati delle transizioni composte da più stati (per es. {q0, q1}), essendo degli insiemi, partendo da quelli precedenti.*
- *Infine, nella scrittura degli stati di transizione degli stati composti, si fa l'unione degli stati visti singolarmente sull'insieme degli stati composti*

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Si nota che l'insieme degli stati, insieme di tutti i possibili sottoinsiemi, cresce esponenzialmente ed il prezzo che si paga nella trasformazione da NFA a DFA è un costo notevole.

La tabella di transizione per D ci permette di ottenere il diagramma di transizione

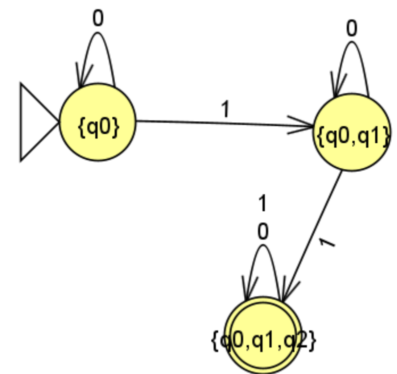


Si nota che alcuni stati possono essere inutili, quindi non raggiungibili dallo stato iniziale.
In questo caso vi sono solo tre stati non raggiungibili e gli altri possono essere omessi.

1. Determinare il DFA equivalente all'NFA con la seguente tabella di transizione:

	0	1
$\rightarrow q_0$	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_1\}$	$\{q_0, q_2\}$
$*q_2$	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$

2. Qual è il linguaggio accettato dall'automa?



1) Qui sopra a destra ho rappresentato l'automa DFA, segue la sua tabella di riferimento (in questo caso completa di tutti i possibili stati, di solito si rappresentano solo gli stati collegati dallo stato iniziale):

Q_d	0	1
Insieme vuoto	Insieme vuoto	Insieme vuoto
{q0}	{q0}	{q0, q1}
{q1}	{q1}	{q0, q2}
*{q2}	{q1, q2}	{q0, q1, q2}
{q0, q1}	{q0, q1}	{q0, q1, q2}
{q0, q1}	{q0, q1, q2}	{q0, q1, q2}
{q0, q1}	{q0, q1}	{q0, q1, q2}
*{q0, q2}	{q0, q1, q2}	{q0, q1, q2}
*{q1, q2}	{q1, q2}	{q0, q1, q2}
*{q0, q1, q2}	{q0, q1, q2}	{q0, q1, q2}

2) Accetta come linguaggio tutte le stringhe che contengono almeno due 1

Nota: l'automa si ottiene, guardando la tabella, per unione delle precedenti. Parto da q0 e interpreto la prima riga. Sono in {q0,q1} e da lì faccio l'unione dei risultati di q0 e q1 (q0 va a 0, q1 va a 0, quindi unisco e q0,q1 andrà a 0 su sé stesso). Poi vado verso q0,q1,q2: questo perché q0 va ad 1 verso q0,q1 mentre q1 va ad 1 con q0,q2. Unisco i due risultati ed ottengo esattamente q0,q1,q2. A questo punto, unisco q0,q1,q2 (avendo q0 che va a q0 per 0, q1 che va a q1 per 0 e q1 che va a 0 per q1,q2) e i loro risultati portano

proprio a q0,q1,q2. Ad 1 si nota che ci sta già una transizione verso q0,q1,q2 quindi non serve metterne altre e sarà quella finale.

Theorem

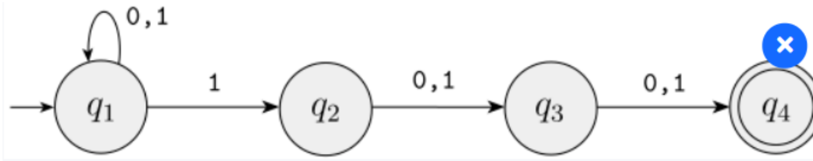
Un linguaggio L è accettato da un DFA se e solo se è accettato da un NFA.

Dimostrazione:

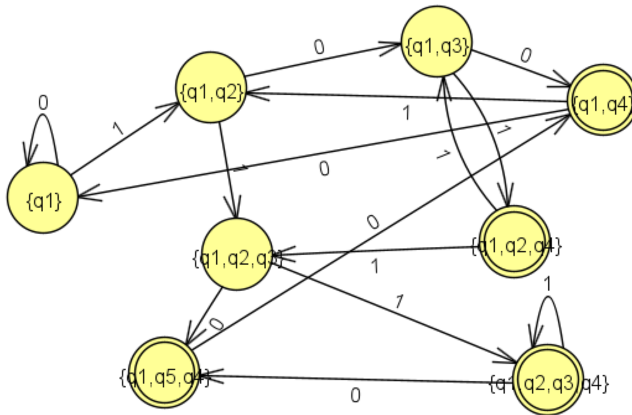
- La parte "se" è data dalla costruzione per sottoinsiemi
- La parte "solo se" si dimostra osservando che ogni DFA può essere trasformato in un NFA modificando δ_D in δ_N con la seguente regola:

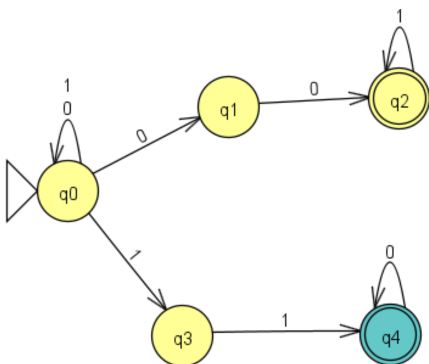
  Se $\delta_D(q, a) = p$ allora $\delta_N(q, a) = \{p\}$

Questo automa fa parte della domanda presente anche sotto che riassumo:
"Prova a costruire un DFA partendo dall'NFA (foto, ndr) che riconosce il linguaggio di tutte le stringhe con un 1 nella terza posizione dalla fine"



Ecco quindi il DFA di riferimento costruito (per unione degli stati precedenti, partendo dallo stato iniziale):





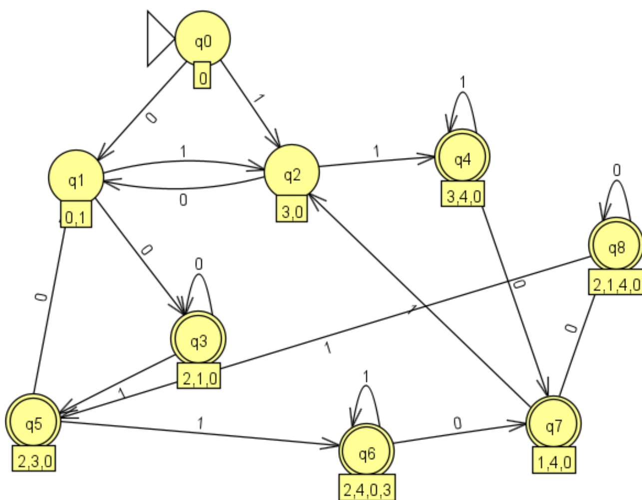
Si può anche usare JFLAP per convertire un NFA ad un DFA e verificare in autonomia se è corretta l'implementazione con tutti gli stati come descritto sopra.
Rappresentiamo su JFLAP l'automa NFA qui a sinistra:

Clicchiamo poi nel menù *Convert* e successivamente *Convert to DFA.*
Nella schermata che si apre *NFA to DFA* avremo un menù di questo tipo in alto, assieme a *Complete* e *Done.*

A parte la selezione si ha:

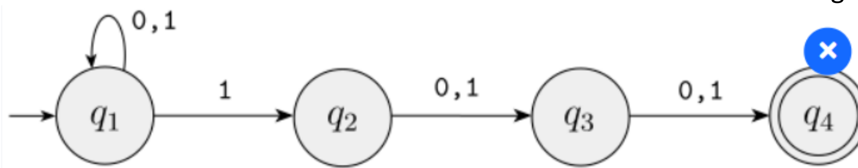
- il secondo pulsante, *Expand Group on Terminal*, da cui si apre una finestra di contesto che chiede di immettere dei valori all'automa specifici per espanderne gli stati e formare tutte le transizioni.
  A questo punto clicco su un automa trascinando fuori con il cursore e sarà chiesto il valore con cui espandere stati/transizioni e l'elenco degli stati su cui operare (scritti senza spazi tra le virgole).
  Se immetto stati errati, JFLAP dirà *The list of states is incorrect.*
- il terzo pulsante, *State Expander*, che permette, cliccando su uno stato, la creazione automatica di transizioni verso gli altri stati esistenti. Fa tutto lui in questo caso.

Cliccando sul pulsante *Done* si verifica se l'NFA corrispondente è stato costruito con tutti gli stati possibili. JFLAP lo segnala aprendo l'automa in un'altra finestra e dicendo *The NFA is fully built*.
Segnalerà altrimenti il numero di transizioni mancanti.
In questo esempio ecco quindi il DFA corrispondente al precedente NFA:



<u>*Domande Wooclap*</u>

1) L'NFA mostrato nella figura riconosce il linguaggio di tutte le stringhe con un 1 nella terza posizione dalla fine. Prova a costruire un DFA che riconosce lo stesso linguaggio: quanti stati possiede?



Risposta: Si risponde costruendo tutte le transizioni da tutte le combinazioni stati quindi *2^4 = 16*
Per capire gli stati effettivamente utili non si fa la tabella di transizione (con cui si trovano tutti gli stati, quindi al completo), ma il diagramma di transizione. Qui gli stati utili sono 8.

2) Qual è l'OUTPUT della funzione di transizione di un NFA?
   Risposta: Un insieme di stati

3) In un NFA, quante transizioni con lo stesso simbolo possiamo avere in uno stato?
   Risposta: Un numero a piacere, zero compreso.

4) Come si rappresenta la computazione di un NFA?
   Risposta: Con un albero di possibili transizioni
5) Un NFA accetta una stringa quando…?
   Risposta: Esiste una computazione che termina in uno stato finale

6) Un NFA rifiuta una stringa quando…?
   Risposta: Nessuna computazione termina in uno stato finale

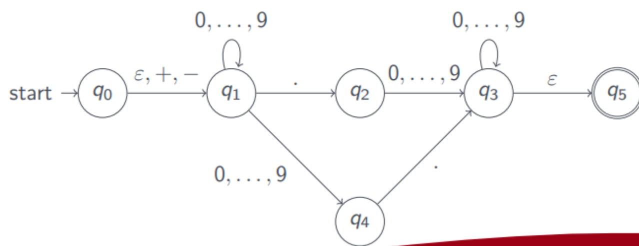
Automi a stati finiti con epsilon-transizioni

Abbiamo quindi automi a stati finiti non deterministici, che sono appunto gli automi ad ε-transizioni.
Iniziamo con un esempio:

Esercizio: costruiamo un NFA che accetta numeri decimali:

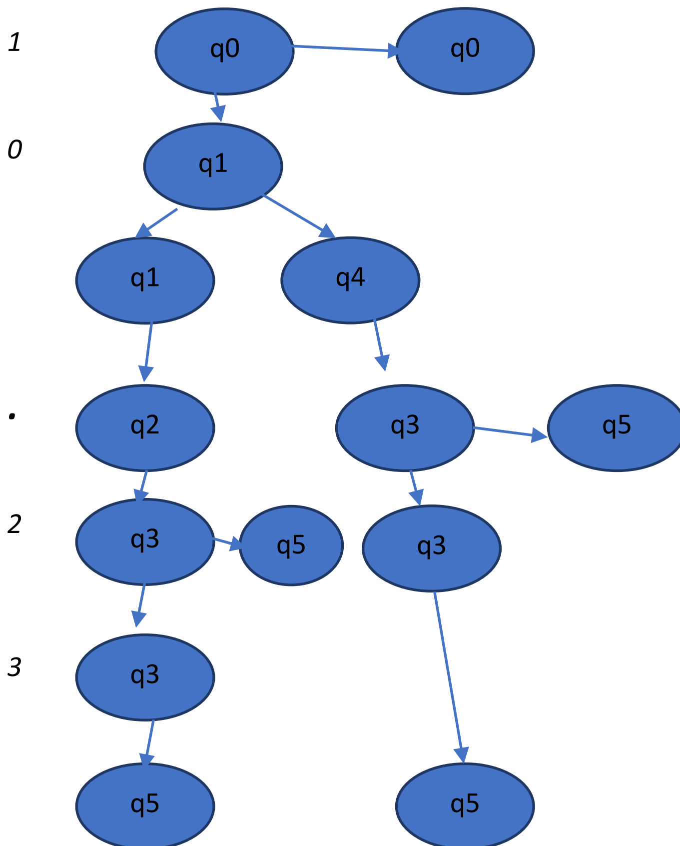
1. Un segno $+$ o $-$, opzionale
2. Una stringa di cifre decimali $\{0, \ldots, 9\}$
3. un punto decimale .
4. un'altra stringa di cifre decimali

Una delle stringhe (2) e (4) può essere vuota, ma non entrambe

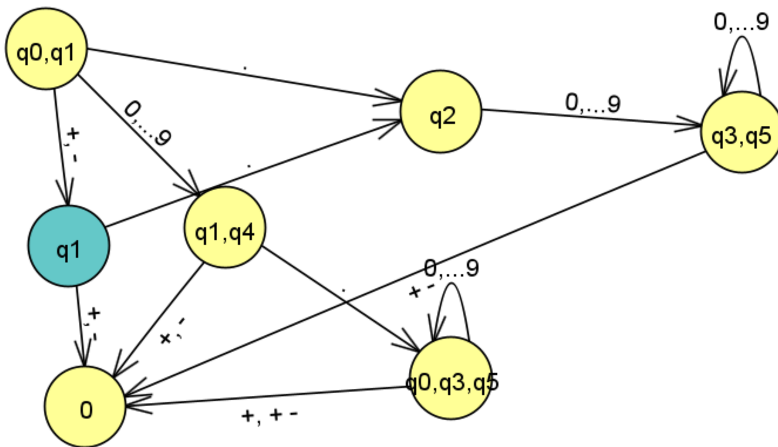


Finora tutti gli automi visti eseguivano una transizione per ogni singolo input; altrimenti possiamo indicare più simboli per non consumare tutto l'input, combinandone vari per una singola transizione.
Nota: se avessi messo nell'immagine sopra q3 come stato finale, poteva andare bene lo stesso.
La computazione che indica che potrebbe non avvenire una transizione viene data da ε.

Qui la rappresentazione ad albero



L'automa chiesto dall'immagine della slide mostrata è il seguente (si noti che nelle frecce da {q0,q1} verso q2 e da {q3,q5} verso lo stato vuoto 0 si ha il punto (.) come simbolo).



In pratica le ε-transizioni permettono di andare oltre i possibili valori presenti non consumando ulteriori transizioni ed eventualmente proseguendo oltre la fine.

Un Automa a Stati Finiti Non Deterministico con ε-transizioni (ε-NFA) è una quintupla

$$A = (Q, \Sigma, \delta, q_0, F)$$

dove:

- Q, Σ, q_0, F sono definiti come al solito
- δ è una funzione di transizione che prende in input:
  - uno stato in Q
  - un simbolo nell'alfabeto $\Sigma \cup \{\varepsilon\}$

  e restituisce un sottoinsieme di Q

L'automa che riconosce le cifre decimali è definito come

$$A = (\{q_0, q_1, \dots, q_5\}, \{+, -, ., 0, \dots, 9\}, \delta, q_0, \{q_5\})$$

dove δ è definita dalla tabella di transizione

	ε	$+, -$	$.$	$0, \dots, 9$
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
$*q_5$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

È possibile eliminare le transizioni, ma noi non lo vedremo adesso; modificheremo invece la conversione da NFA ad un DFA per poi realizzare una trasformazione da ε-NFA a ε-DFA.
L'eliminazione all'interno del DFA degli stati raggiungibili/possibili rappresenta l'insieme degli stati ottenibili con le operazioni date. Calcoliamo quindi l'insieme delle ε-transizioni, per poi effettuarne l'eliminazione induttivamente. Questa operazione viene definita come *ε-chiusura*.

L'eliminazione delle ε-transizioni procede per ε-chiusura degli stati:

- tutti gli stati raggiungibili da q con una sequenza $\varepsilon\varepsilon\dots\varepsilon$

La definizione di $\mathrm{ECLOSE}(q)$ è per induzione:

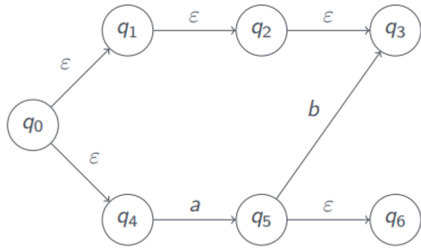
Caso base:

$$q \in \mathrm{ECLOSE}(q)$$

Caso induttivo:

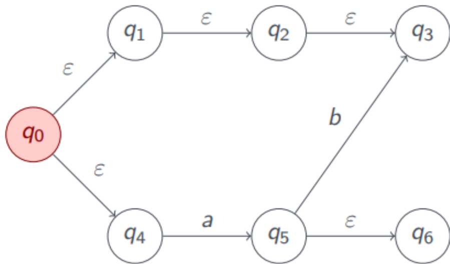
se $p \in \mathrm{ECLOSE}(q)$ e $r \in \delta(p, \varepsilon)$ allora $r \in \mathrm{ECLOSE}(q)$

Calcoliamo quindi induttivamente la ε-chiusura partendo da:



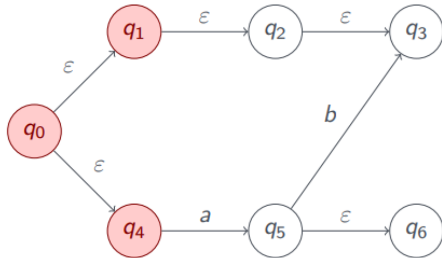
$\mathrm{ECLOSE}(q_0) = \{$

Ad esempio, partendo da q0, avremo:



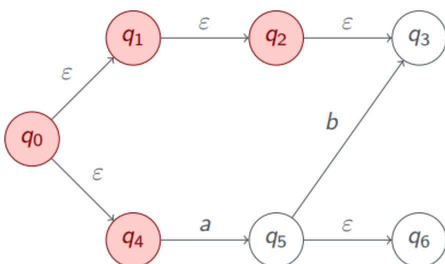
$\mathrm{ECLOSE}(q_0) = \{q_0$

Seguono poi le altre possibili:



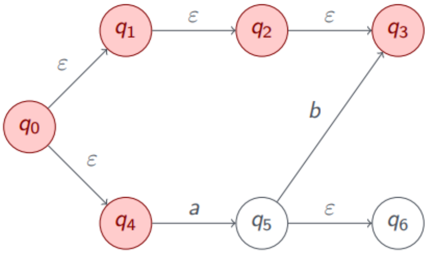
$\mathrm{ECLOSE}(q_0) = \{q_0, q_1, q_4$

Verifico poi se posso applicare il caso induttivo su q5 e q2; quindi proseguo solo su q2, perché dalla tabella di transizione sopra si vede che su q5 muore (dead state):

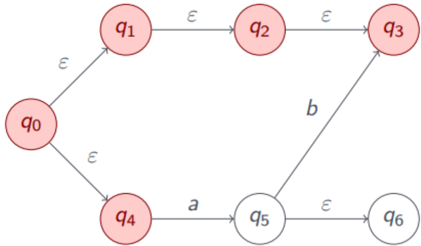


$\mathrm{ECLOSE}(q_0) = \{q_0, q_1, q_4, q_2$

E così via quindi, inoltre concludendo:



$\mathrm{ECLOSE}(q_0) = \{q_0, q_1, q_4, q_2, q_3$



$\mathrm{ECLOSE}(q_0) = \{q_0, q_1, q_4, q_2, q_3\}$

- Anche in questo caso abbiamo definito una classe di automi che è equivalente ai DFA
- Per ogni ε-NFA E c'è un DFA D tale che $L(E) = L(D)$, e viceversa
- Lo si dimostra modificando la costruzione a sottoinsiemi: Dato un ε-NFA

$$E = (Q_E, \Sigma, q_0, \delta_E, F_E)$$

costruiremo un DFA

$$D = (Q_D, \Sigma, S_0, \delta_D, F_D)$$

tale che

$$L(D) = L(E)$$

Definiamo poi come sempre i membri matematicamente, prendendo quindi in sintesi ogni singola ε-transizione, realizzandone poi la chiusura e rappresentando gli stati finali, qualora ci sia almeno uno stato finale, realizzando quindi ogni singolo stato e anche quelli raggiungibili.
Anche in questo caso l'insieme degli stati possibili risulta essere esponenziale.

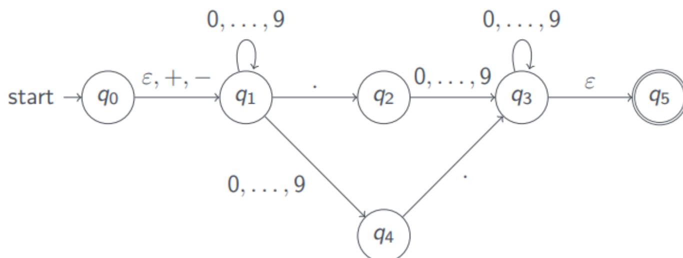
- $Q_D = \{S \subseteq Q_E : S = \text{ECLOSE}(S)\}$
  Ogni stato è un insieme di stati chiuso per ε-transizioni
- $S_0 = \text{ECLOSE}(q_0)$
  Lo stato iniziale è la ε-chiusura dello stato iniziale di E
- $F_D = \{S \in Q_D : S \cap F_E \neq \emptyset\}$
  Uno stato del DFA è finale se c'è almeno uno stato finale di E
- Per ogni $S \in Q_D$ e per ogni $a \in \Sigma$:

$$\delta_D(S, a) = \text{ECLOSE}\Big(\bigcup_{p \in S} \delta_E(p, a)\Big)$$

  La funzione di transizione "percorre tutte le possibili strade" (comprese quelle con ε-transizioni)

Andiamo quindi ad un esempio pratico:

Costruiamo un DFA D equivalente all'ε-NFA E che riconosce i numeri decimali:



Si applicano le regole delle ε-chiusure, descrivendo quali sono gli stati iniziali, considerando anche lo stato stesso q0 nel primo caso e poi prendendo più stati nel caso in cui si abbia ε tra questi.

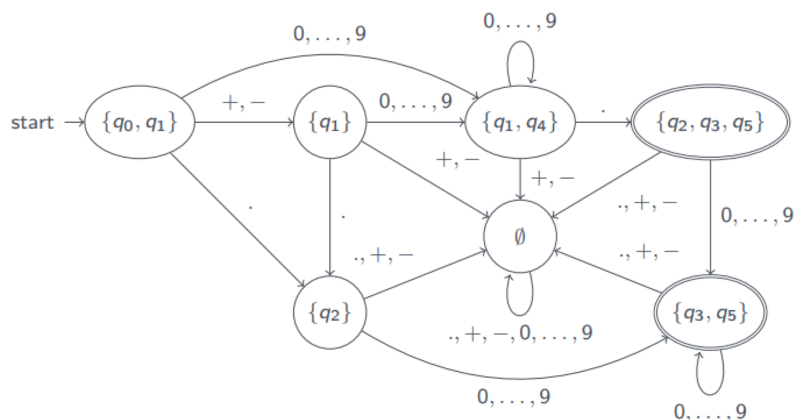
- Come prima cosa costruiamo la ε-chiusura di ogni stato:

$$\begin{aligned} \text{ECLOSE}(q_0) &= \{q_0, q_1\} & \text{ECLOSE}(q_1) &= \{q_1\} \\ \text{ECLOSE}(q_2) &= \{q_2\} & \text{ECLOSE}(q_3) &= \{q_3, q_5\} \\ \text{ECLOSE}(q_4) &= \{q_4\} & \text{ECLOSE}(q_5) &= \{q_5\} \end{aligned}$$

- Lo stato iniziale di D è $\{q_0, q_1\}$

Per arrivare a questo automa usamo la tabella presente a pagina 15. La ε-chiusura mi permette di stabilire lo stato iniziale utile (quindi q0,q1) e poi da quello sviluppare tutti gli altri stati per mezzo della tabella citata (generalmente si prende come stato iniziale la chiusura più grande e che comprende anche lo stato iniziale). Gli stati unione anche di stati precedenti sono dati dalle ε, dove si può notare che q0 è ε di q1, da cui sono uniti nello stato iniziale,

- Applicando le regole otteniamo il diagramma di transizione:



similmente anche q3 con q5. q1,q4,q5 si forma dal fatto di avere q5 come stato finale unico. Consegue poi dalla costruzione a sottoinsiemi tutto l'automa (usando come al solito gli stati utili).
Si noti che negli esercizi, *tutti gli stati che con una ε-transizione fanno parte di una ε-chiusura, vanno necessariamente considerati nella costruzione dell'automa*. Questo è un passaggio chiaro facendo esercizio (in fondo al file).

Theorem

Un linguaggio L è accettato da un DFA se e solo se è accettato da un ε-NFA.

Dimostrazione:

- La parte "se" è data dalla costruzione per sottoinsiemi modificata
- La parte "solo se" si dimostra osservando che ogni DFA può essere trasformato in un ε-NFA modificando δ_D in δ_E con la seguente regola:

  Se $\delta_D(q, a) = p$ allora $\delta_E(q, a) = \{p\}$

Le espressioni sui linguaggi regolari prevedono una serie di operazioni insiemistiche, descritte di fianco:

Nota importante:
L'unione viene scritta anche come "pipe", cioè con il segno "|", quindi:
(A|B) è equivalente a (A+B), che rappresentano appunto l'unione dei due stati A e B.
Per esempio, su Automata Tutor viene usato "|".

- **Unione:**
  $$L \cup M = \{w : w \in L \text{ oppure } w \in M\}$$
- **Intersezione:**
  $$L \cup M = \{w : w \in L \text{ e } w \in M\}$$
- **Concatenazione:**
  $$L.M = \{uv : u \in L \text{ e } v \in M\}$$
- **Complemento:**
  $$\overline{L} = \{w : w \notin L\}$$
- **Chiusura (o Star) di Kleene:**
  $$L^* = \{w_1 w_2 \dots w_k : k \geq 0 \text{ e ogni } w_i \in L\}$$

<u>*Domande Wooclap*</u>

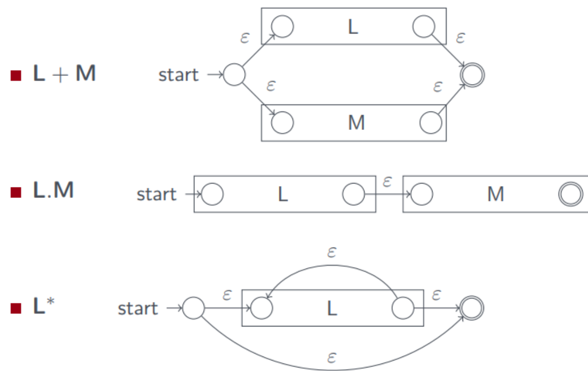
1) Una epsilon-transizione è:
   Risposta: Una transizione che non consuma l'input

2) NFA ed ε-NFA riconoscono la stessa classe di linguaggi?
   Risposta: Vero

3) Siano *L* e *M* linguaggi regolari. Per ognuna delle operazioni seguenti, indica se puoi costruire un automa che riconosce il linguaggio. Puoi scegliere più di una risposta.
   Risposte (qui a destra listate):

   - Unione $L \cup M$
   - Concatenazione $L.M$
   - Star L^*
   - Intersezione $L \cap M$

Ecco esempi possibili di automi (risposta quindi anche al file di esercizi “*03-esercizi*”).



Linguaggi regolari

Partiamo dall’insieme dalle 5 operazioni dell’altra volta.
Con esse, sono definite anche le proprietà di *chiusura* di questi linguaggi:

Se L e M sono linguaggi regolari, allora anche i seguenti linguaggi sono regolari:

- Unione: $L \cup M$
- Intersezione: $L \cap M$
- Concatenazione: $L.M$
- Complemento: $\overline{L}$
- Chiusura di Kleene: L^*

Ciò è dimostrabile per ogni singola operazione, ad esempio nel caso dell’intersezione:

Se L e M sono regolari, allora anche $L \cap M$ è un linguaggio regolare.

Dimostrazione. Sia L il linguaggio di

$$A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$$

e M il linguaggio di

$$A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$$

Possiamo assumere che entrambi gli automi siano deterministici
Costruiremo un automa che simula A_L e A_M in parallelo, e accetta se e solo se sia A_L che A_M accettano.

In questo caso specifico ammettiamo che entrambi siano DFA e quindi entrambi accettino l’intersezione (in questo caso quindi appartiene sia al linguaggio L che al linguaggio M), eseguendo in parallelo i due automi. Si sfrutta il determinismo per dire che la coppia esiste ed è unica e il risultato dell’intersezione corrisponde allo stato della coppia dopo aver letto la parola, mettendo insieme entrambi i pezzi:

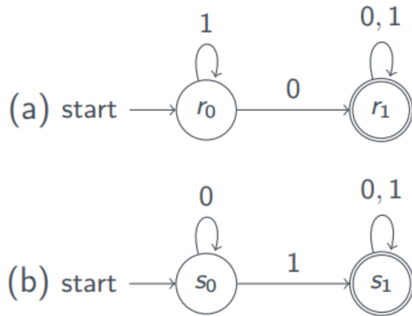
Dimostrazione (continua).
Se A_L va dallo stato p allo stato s leggendo a, e A_M va dallo stato q allo stato t leggendo a, allora $A_{L\cap M}$ andrà dallo stato (p,q) allo stato (s,t) leggendo a.
Formalmente

$$A_{L\cap M} = (Q_L \times Q_M, \Sigma, \delta_{L\cap M}, (q_L, q_M), F_L \times F_M),$$

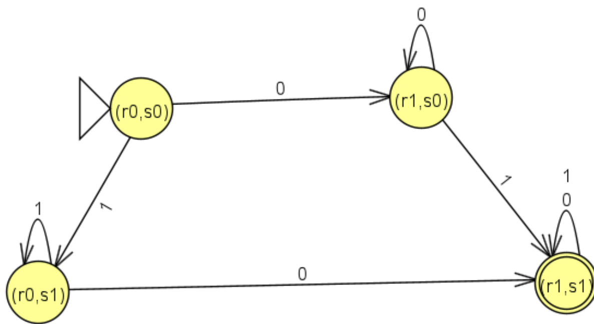
dove

$$\delta_{L\cap M}((p,q), a) = (\delta_L(p,a), \delta_M(q,a))$$

Costruiamo per esempio l'automa che rappresenta l'intersezione dei seguenti:



Questo sotto è il DFA risultante; di fatto basta letteralmente considerare tutti i casi.
Ad esempio, con {r0,s0}, avremo che r0 va ad r1 per 0, mentre s0 va a sé stesso con 0, quindi lo stato risultante sarà {r1.s0} per questi motivi. Seguono poi tutti gli altri stati:



Parliamo ora di ***espressioni regolari***, quindi dei modi dichiarativi per scrivere un linguaggio regolare, in particolare riconoscendo tutti i linguaggi dotati di operazioni di chiusura, che hanno un insieme finito di stringhe (*teorema di Kleene*).
Queste possono essere scritte tramite editor di testo, comandi UNIX, strumenti di analisi lessicale.
Si può usare un FA (NFA/DFA) per costruire una macchina che riconosce linguaggi regolari.
Esempio di linguaggio regolare: 01* + 10*

Esse vengono costruite utilizzando:

- un insieme di **costanti** di base:
  - ε per la stringa vuota
  - $\emptyset$ per il linguaggio vuoto
  - a, b, ... per i simboli $a, b, \dots \in \Sigma$
- collegati da **operatori**:
  - $+$ per l'unione
  - $\cdot$ per la concatenazione
  - $*$ per la chiusura di Kleene
- raggruppati usando le **parentesi**:
  - ( )

- $L(\varepsilon) = \{\varepsilon\}$
- $L(\emptyset) = \emptyset$
- $L(\mathrm{a}) = \{a\}$

Induttivamente, il linguaggio L(E) viene rappresentato con un caso base:

Il caso induttivo poi individua i possibili linguaggi rappresentati avendo un operatore:
(con la * che rappresenta la chiusura di Kleene e le parentesi che rappresentano l'unione).

- $L(E + F) = L(E) \cup L(F)$
- $L(EF) = L(E).L(F)$
- $L(E^*) = L(E)^*$
- $L((E)) = L(E)$

Primo esempio:

- Scriviamo l'espressione regolare per

$$L = \{w \in \{0,1\}^* : 0 \text{ e } 1 \text{ alternati in } w\}$$

Fanno parte del linguaggio parole come: 0101, 101, 101010….
In questo caso quindi ragioniamo su tutti gli input di parole possibili, quindi nell'ordine:
caso in cui ripete solo 01, caso in cui si ripete 10, caso in cui si ripete solo 01 preceduto da 1 e caso in cui si ripete solo 10 preceduto da 0.

Soluzione:
(01)* + (10)* + 1(01)* + 0(10)*
oppure in maniera compatta:
(ε+1)(01)*(ε+0)

Similmente alle operazioni aritmetiche, anche qui dobbiamo adoperare delle *regole di precedenza* nel caso degli operatori, denotando linguaggi diversi.

1. Chiusura di Kleene
2. Concatenazione (punto)
3. Unione (+)

Esempio:

$$01^* + 1 \text{ è raggruppato in } (0(1)^*) + 1$$

e denota un linguaggio diverso da

$$(01)^* + 1$$

Piccola nota:
Noi non abbiamo detto questa cosa, tuttavia:
quando noi usiamo l'operatore (*) allora abbiamo tutte le combinazioni possibili di un certo carattere o di una sottostringa, includendo anche la stringa vuota.
Quando esplicitamente si ha una (*) che non include la stringa vuota, eleviamo a (+) invece che a (*).

Ad esempio:
{ε, 0, 00, 000, …) in questo caso è chiusura su 0, e quindi scrivo 0*
Ma se io avessi {0,00,000…} qui non si ha la stringa vuota e scriverò 0^+

<u>*Domande Wooclap*</u>:

Sappiamo che R e S sono due automi a stati finiti.
Associate ogni figura al linguaggio riconosciuto dall'automa.

In queste risposte sono nell'ordine: concatenazione (sx), unione (dx) e chiusura di Kleene/star (sotto).

A. $L(R).L(S)$

✅ A. $L(R).L(S)$

C. $L(R)^*$

✅ B. $L(R) \cup L(S)$

B. $L(R) \cup L(S)$

✅ C. $L(R)^*$

Supponi di modificare la costruzione che abbiamo usato per fare l'unione di linguaggi regolari in modo che produca l'automa

$$A = (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), F)$$

dove

$$\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a)) \text{ (come prima)}$$

ma

$$F = \{(p, q) \mid p \in F_1 \text{ o } q \in F_2\},$$

cioè F è l'insieme delle coppie dove l'uno o l'altro elemento è uno stato finale.

Cosa ci darebbe questa costruzione?

Risposta: Unione L U M

Teorema: Se L è un linguaggio regolare, allora anche $\overline{L}$ è un linguaggio regolare.

DIMOSTRAZIONE:

Sia $A = (Q, \Sigma, \delta, q_0, F)$ un NFA che accetta il linguaggio L. Allora se scambio stati finali con stati non finali ottengo l'NFA $\overline{A} = (Q, \Sigma, \delta, q_0, Q - F)$ che riconosce il linguaggio $\overline{L}$

DOMANDA: è giusta questa dimostrazione?

Risposta:
L'automa è nondeterministico, rappresentabile da un albero di computazione.
In questo caso, scambiare stati finali con non finali funziona lo stesso; l'osservazione funziona solo se abbiamo un DFA con una sola computazione (rappresentabile una lista linkata).
Infatti un DFA sceglie una sola strada per finire; NFA ha il nondeterminismo, quindi può scegliere più strade e non è più verificata la proprietà di univocità. Ragionevolmente quindi se la domanda fosse stata rivolta ad un DFA, la risposta sarebbe stata SI per questo motivo.

Risposta corretta: NO, non è corretta

<u>Quali espressioni regolari possono generare la stringa "ababout"? Puoi scegliere più di una risposta.</u>
Risposte:
- (ab)*out
- (a + b)*out

<u>Quali espressioni regolari possono generare la stringa "abbbout"? Puoi scegliere più di una risposta.</u>
Risposte:
- (ab)*out
- (a + b)*out

<u>Quali espressioni regolari possono generare la stringa "bbbbbbout"? Puoi scegliere più di una risposta.</u>
Risposta:
- (a+b)*out

<u>Quali espressioni regolari possono generare la stringa "out"? Puoi scegliere più di una risposta.</u>
Risposte:
- (ab)*out
- (ab)* + out
- (a + b)*out

Costruisci una ER sull'alfabeto $\{0, 1\}$ che rappresenti tutte le stringhe che contengono la sottostringa 101

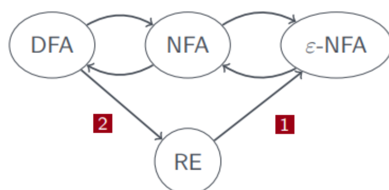
Risposte:
- (0+1)*101(0+1)*
- (0+1)*(101)(0+1)*

Costruisci una ER sull'alfabeto $\{a, b, c\}$ che rappresenti tutte le stringhe la cui lunghezza è un multiplo di 3

Risposta:
- ((A+B+C)(A+B+C)(A+B+C))*

Equivalenza tra FA e RE e conversione per eliminazione di stati

Sappiamo già che DFA, NFA, e ε-NFA sono tutti equivalenti.



Gli FA sono equivalenti alle espressioni regolari:

1. Per ogni espressione regolare $\mathbf{R}$ esiste un ε-NFA A, tale che $L(A) = L(\mathbf{R})$
2. Per ogni DFA A possiamo costruire un'espressione regolare $\mathbf{R}$, tale che $L(\mathbf{R}) = L(A)$

Similmente, possiamo costruire un ε-NFA A tale che L(A) = L(R).

Le restrizioni imposte semplificano la costruzione finale, spezzando tutto nelle sottoespressioni induttivamente:

Dimostrazione:

Costruiremo un ε-NFA A con:

- un solo stato finale
- nessuna transizione entrante nello stato iniziale
- nessuna transizione uscente dallo stato finale

La dimostrazione è per induzione strutturale su **R**

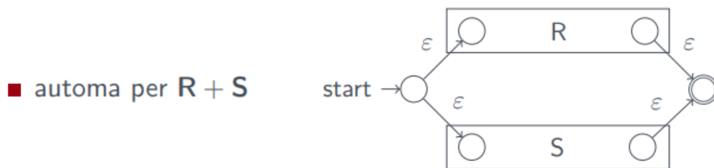
Come caso base si pone l'automa che va in uno stato finale partendo da ε, in cui non può fare nulla, rifiutando la parola.

Caso Base:

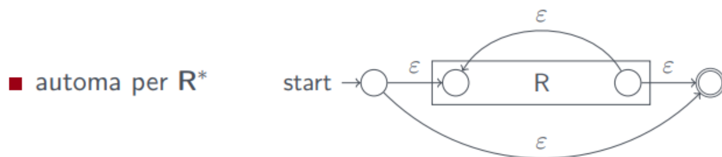
Similmente per il linguaggio vuoto che non contiene nessuna parola, rifiuta tutte le parole, compresa quella vuota:

Così, anche nel caso di una lettera, abbiamo un'espressione regolare che termina subito, come nel caso:

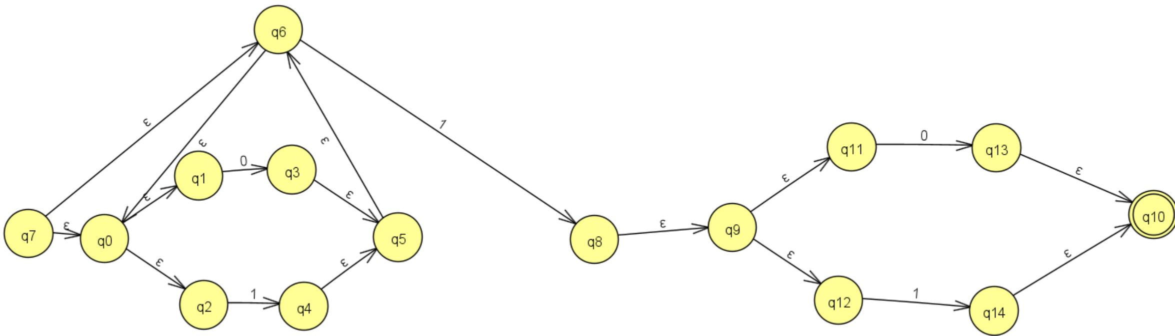
Induttivamente, possiamo costruire varie combinazioni di automa, come quello della somma, facendo l'unione:



Similmente aggiungiamo sia l'automa di concatenazione (RS) che quello della chiusura di Kleene (R*).

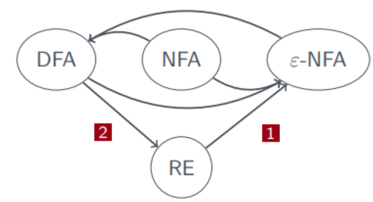


Vediamo quindi un esempio:
Trasformazione dell'espressione (0 + 1)∗1(0 + 1) in ε-NFA



Inoltre è possibile operare una conversione di un qualsiasi automa in una espressione regolare equivalente, procedendo per *eliminazione degli stati*.
I cammini quindi scompaiono, aggiungendo nuove transizioni etichettate con espressioni regolari.
Elimina quindi gli stati sostituendo tutto con altre espressioni, una volta eliminate tutte le transizioni.
Induttivamente ottiene sempre un'espressione regolare (per chiusura).

Sappiamo già che DFA, NFA, e ε-NFA sono tutti equivalenti.

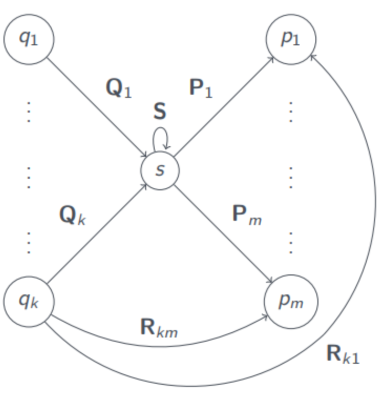


Gli FA sono equivalenti alle espressioni regolari:

1. Per ogni espressione regolare $\mathbf{R}$ esiste un ε-NFA A, tale che $L(A) = L(\mathbf{R})$
2. Per ogni FA A possiamo costruire un'espressione regolare $\mathbf{R}$, tale che $L(\mathbf{R}) = L(A)$

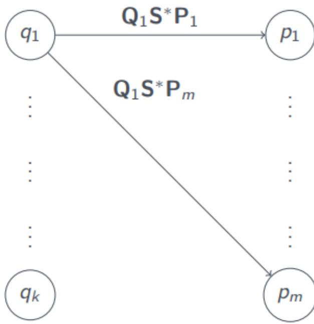
- La procedura che vedremo è in grado di convertire un automa (DFA, NFA, ε-NFA) in una espressione regola equivalente
- Si procede per eliminazione di stati
- Quando uno stato q viene eliminato, i cammini che p per q scompaiono
- si aggiungono nuove transizioni etichettate con espres regolari che rappresentano i cammini eliminati
- alla fine otteniamo un'espressione regolare che rappre tutti i cammini dallo stato iniziale ad uno stato finale
  - $\Rightarrow$ cioè il linguaggio riconosciuto dall'automa

Quindi vogliamo lavorare di eliminazione dello stato (utile è capire tutte le dipendenze e tutti i percorsi che passano o raggiungono lo stato che vogliamo eliminare in quel momento):

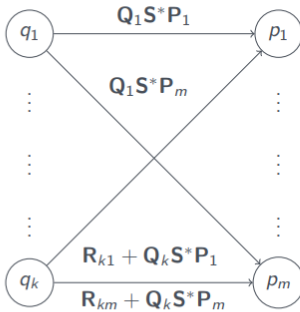


- Lo stato da eliminare può avere un ciclo
- $q_1, \dots, q_k$ sono i predecessori
- $p_1, \dots, p_m$ sono i successori
- ci possono essere transizioni dirette tra i predecessori ed i successori

Occorre quindi ricreare la transizione per ogni coppia predecessore/successore ad esempio così:



Se non ci sta transizione diretta, l'etichetta è $Q_i S{*}P_j$ (concatenazione)
Se invece c'è una transizione diretta si ha l'unione (R_{ij} + $Q_iS{*}P_j$), come nel caso sottostante:

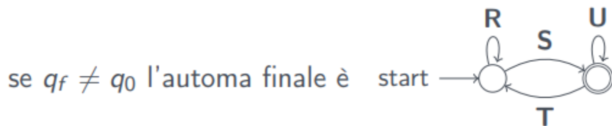


<u>Attenzione che:</u>

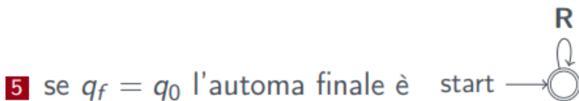
- se lo stato iniziale presenta più stati entranti verso lo stato iniziale occorre crearne uno nuovo iniziale che non ha archi entranti, aggiungendo una ε-transizione verso il nuovo stato iniziale da parte del vecchio stato iniziale;
- se l'automa presenta più stati finali oppure presenta più transizioni uscenti dallo stato finale, si convertono tutti gli stati finali in stati non-finali, creando un nuovo stato finale e aggiungendo una ε-transizione verso il nuovo stato finale da parte dei vecchio stati finali.

La strategia completa immagina che:

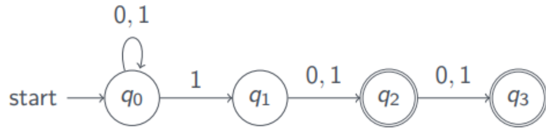
1) l'automa di partenza abbia un unico stato finale (avendone più di uno ne crea uno nuovo con ε-transizioni provenienti da vecchi stati)
2) collassa le transizioni, in cui una serie di transizioni può essere etichettata come una singola (es. a + b) mettendo insieme tutte le possibilità tra la stessa coppia di stati
3) elimina tutti gli stati tranne lo stato iniziale e lo stato finale (in qualsiasi ordine)



Altro caso, più banale (caso iniziale che coincide con lo stato finale):



Costruiamo l'espressione regolare equivalente al seguente NFA:



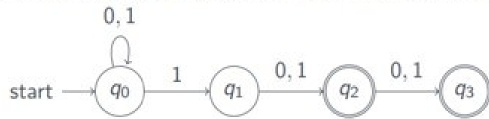
Alcune note pratiche (segue l'espressione equivalente sotto):

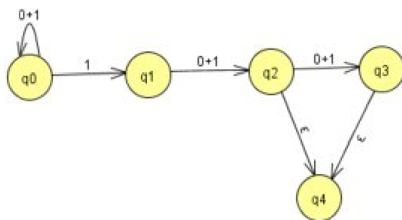
- L'unione, da quello che ho capito, è rappresentata dal fatto di avere uno stato che può avere più transizioni uscenti, verso sé stesso, oppure verso altre transizioni.
- Un ciclo è rappresentato da almeno uno stato di ritorno (quindi un'espressione complessa cicla su sé stessa avendo delle frecce di ritorno precedenti)
- Una concatenazione è rappresentata da frecce uscenti consecutive
- Il ciclo sullo stato iniziale non si conta come ciclo all'inizio delle transizioni quando scrivo la ER (quando sto esaminando lo stato iniziale) ma si conta come ciclo quando lo vedo come stato intermedio (se non si capisce cosa intendo, si faccia un esercizio e si capisce bene così come è scritta).
- Una transizione che va avanti e una che va indietro tra due stati si considera concatenazione

La costruzione equivalente è:

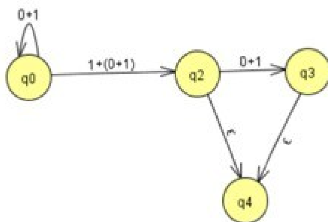
Esercizi

Costruiamo l'espressione regolare equivalente al seguente NFA:



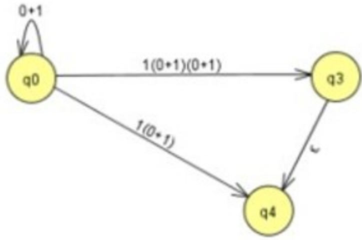


Elimino q1 **PRE: q0** **SUCC: q2** **1(0+1)**

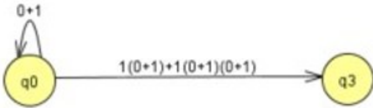


Elimino q2 **PRE: q0** **SUCC: q3** **1(0+1)(0+1)**

PRE: q0 **SUCC: q4** **1(0+1)**

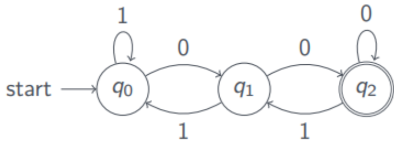


Elimino q3 **PRE: q0** **SUCC: q4** **1(0+1)(0+1)**



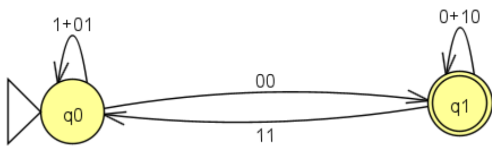
L'espressione finale è:
(0+1)*(1(0+1)+1(0+1)(0+1))

2 Costruiamo l'espressione regolare equivalente al seguente NFA:



	PRE	SUCC	
Elimino q1	q0	q0	1+01
	q0	q2	00
	q2	q2	0+10
	q2	q0	11

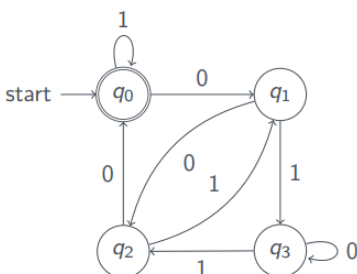
Segue:



ER Finale: ((1+01)+00(0+10)*11)*00(0+10)*

Ultimo esercizio della slide *03-regexp* (tra l'altro nella pag. seguente è una delle domande interattive, comunque):

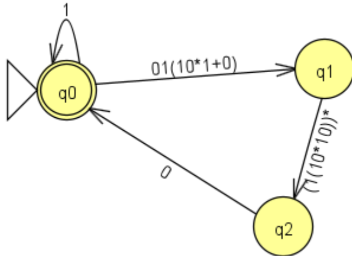
Costruiamo l'espressione regolare equivalente al seguente NFA:



Qui è un macello; comincio con togliere q1

Pre	Post	
q0	q3	0(10*1+0)
q0	q2	01(10*1+0)
q2	q3	1(10*1+0)

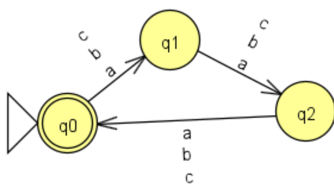
Quindi avremmo:



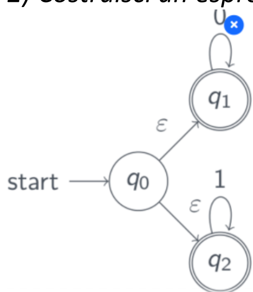
A questo punto, togliamo q3 e avremmo:
1+0(10*1+0)
e se togliamo q2 avremmo l'altro pezzo, quindi
(1(10*1+0))*0
dove abbiamo lo star ulteriore vedendo che prima in q1 c'è il doppio ciclo e gli 0/1 che vanno nella stessa direzione in self-loop.
L'espressione finale è:
(1+0(10*1+0)(1(10*1+0))*0)*

<u>Domande Wooclap</u>

1) *Costruisci un automa equivalente all'espressione regolare ((a+b+c)(a+b+c)(a+b+c))* (puoi caricare un'immagine con la soluzione)*

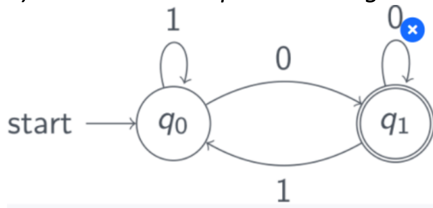


2) Costruisci un'espressione regolare equivalente all'automa in figura:



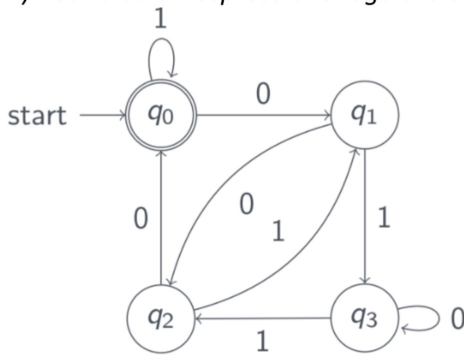
0*+1*

3) Costruisci un'espressione regolare equivalente all'automa in figura:



10(1*00*)*

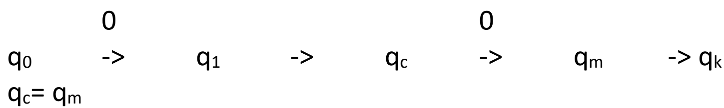
4) Costruisci un'espressione regolare equivalente all'automa in figura:



(1+0(10*1+0)(1(10*1+0))*0)*

Linguaggi non regolari

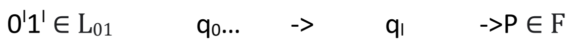
Per dimostrare che non è vero che $L_{01} = \{0^n1^n : n \geq 0\}$ non sia regolare, usiamo la dimostrazione per assurdo. Allora esiste un DFA A che riconosce L_{01}. A è composto da K stati. Diamo in input 0^i con i > k.



$q_0 \xrightarrow{0} q_1 \rightarrow q_c \xrightarrow{0} q_m \rightarrow q_k$

$q_c = q_m$

Il numero di 0 non è limitato, la macchina dovrà tenere traccia di un numero illimitato di possibilità, dato che un DFA ha bisogno di memorizzarsi tutti i passaggi, ma con stati finiti, ovviamente, non è possibile.

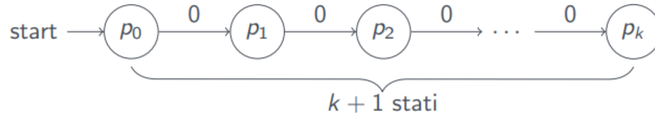


$0^l1^l \in L_{01}$ $\quad q_0\ldots \rightarrow q_l \rightarrow P \in F$

$0^m1^l \in L_{01}$ $\quad q_0\ldots \rightarrow q_l \rightarrow q_m \rightarrow P \in F$

L'automa quindi si ritrova costretto ad accettare qualsiasi stato possibile, perché una delle parti che consuma uno dei pezzi si trova nello stesso stato anche dopo aver ricevuto la nuova transizione (si ha un ciclo quindi). Ciò rappresenta una contraddizione. Pertanto il linguaggio non è regolare. Questa data a parole dal prof è seguita da una più chiara spiegazione dalle slide.

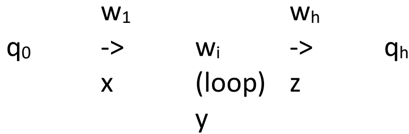
- Supponiamo che $L_{01} = \{0^n 1^n : n \geq 0\}$ sia regolare
- Allora deve essere accettato da un DFA A con un certo numero k di stati
- Cosa succede quando A legge 0^k?
- Seguirà una qualche sequenza di transizioni:



- Siccome ci sono $k + 1$ stati nella sequenza, esiste uno stato che si ripete: esistono $i < j$ tali che $p_i = p_j$
- Chiamiamo q questo stato

- Cosa succede quando l'automa A legge 1^i partendo da q?
- Se l'automa finisce la lettura in uno stato finale:
  - allora accetta, sbagliando, la parola $0^j 1^i$
- Se l'automa finisce la lettura in uno stato non finale:
  - allora rifiuta, sbagliando, la parola $0^i 1^i$
- In entrambi i casi abbiamo ingannato l'automa, quindi L_{01} non può essere regolare

Prendendo per esempio una parola |w| >= k, dove "k" è il numero di stati.
|w|=h



Partendo da questo, posso costruire un numero infinito di parole simili a quella di partenza.
Ad esempio:

$xz \in L(A)$ $\quad$ $xyyz \in L(A)$ $\quad$ $xyyyz \in L(A)$

$xy^iz \in L(A)$ $\quad$ $\forall i >= 0$

Introduciamo quindi il ***pumping lemma***, che afferma che una parola possa essere spezzata in tre parti valendo generalmente nel seguente modo:

Theorem (Pumping Lemma per Linguaggi Regolari)

Sia L un linguaggio regolare. Allora

- *esiste una lunghezza $k \geq 0$ tale che*
- *ogni parola $w \in L$ di lunghezza $|w| \geq k$*
- *può essere spezzata in $w = xyz$ tale che:*
  1. *$y \neq \varepsilon$ (il secondo pezzo è non vuoto)*
  2. *$|xy| \leq k$ (i primi due pezzi sono lunghi al max k)*
  3. *$\forall i \geq 0$, $xy^iz \in L$ (possiamo "pompare" y rimanendo in L)*

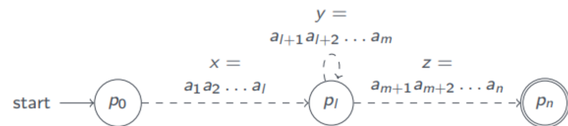
In generale, il pumping lemma afferma che tutti i linguaggi debbano contenere almeno la lunghezza di tutte le stringhe entro un certo valore speciale, chiamato ***pumping length***. Se un linguaggio non rispetta questa proprietà, allora, non è regolare.

Questa proprietà vale perché, considerando tutti gli stati della computazione, almeno uno stato si ripete. Supponendo un linguaggio regolare L, riconosciuto da un DFA con circa *k* stati, consideriamo poi una parola $w=a_1,a_2\ldots a_n$ di lunghezza n >= k.

Definendo così gli stati di tutta la computazione in A, in essi ne esisterà uno che si ripete, per esempio p_l
p_0 -> p_1 -> p_l p_m -> p_n
Esistono quindi: $l < m$ tali che $p_l = p_m$ e $m \leq k$
Iterando nel loop ogni parola $xy^iz \in L(A)$, $\forall i >= 0$

- Possiamo spezzare w in tre parti $w = xyz$:
  1. $x = a_1 a_2 \ldots a_l$
  2. $y = a_{l+1} a_{l+2} \ldots a_m$
  3. $z = a_{m+1} a_{m+2} \ldots a_n$
- che rispettano le condizioni del Lemma:
  - $y \neq \varepsilon$ perché $l < m$
  - $|xy| \leq k$ perché $m \leq k$

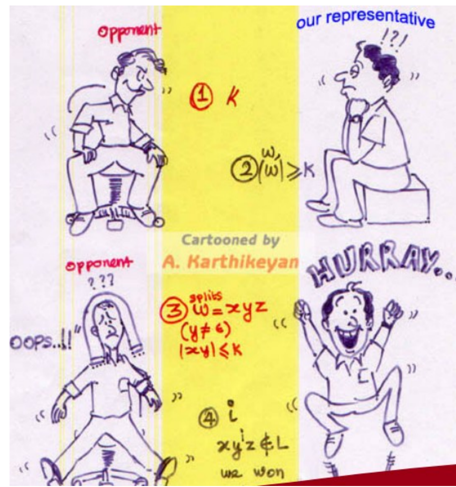
- Quindi, nel grafo delle transizioni di A:



- E di conseguenza anche xy^iz viene riconosciuta dall'automa per ogni $i \geq 0$

- Ogni linguaggio regolare soddisfa il Pumping Lemma.
- Un linguaggio che falsifica il Pumping Lemma non può essere regolare:
  - per ogni lunghezza $k \geq 0$
  - esiste una parola $w \in L$ di lunghezza $|w| \geq k$ tale che
  - per ogni suddivisione $w = xyz$ tale che:
    1. $y \neq \varepsilon$ (il secondo pezzo è non vuoto)
    2. $|xy| \leq k$ (i primi due pezzi sono lunghi al max k)
  - esiste un $i \geq 0$ tale che $xy^iz \notin L$ (possiamo "pompare" y ed uscire da L)
- Attenzione: esistono linguaggi non regolari che rispettano il Pumping Lemma!

Il Pumping Lemma quindi considera non regolare i linguaggi che non rispettano le condizioni di stringa pompata. Un linguaggio può essere non regolare, ma rispettare le condizioni di PL. Vediamo quindi l'idea del ***gioco del pumping lemma*** (giocatore 2 deve dimostrare che il linguaggio non è regolare (noi), mentre il giocatore 1 deve dimostrare che il linguaggio è regolare):



- L'avversario sceglie la lunghezza k
- Noi scegliamo una parola w
- L'avversario spezza w in xy
- Noi scegliamo tale che $xy^iz \notin L$
- allora abbiamo vinto

È possibile fare la stessa cosa sul mitico JFLAP.
Si può scegliere se lasciare il computer o se lasciare il controllo al giocatore.
Si seleziona "*Regular Pumping Lemma*"
Per esempio sulla espressione di prima, selezionando "*Computer goes first*", dimostriamo che una stringa pompata non fa parte del linguaggio (quindi non rispetta la condizione di prima):

$L = \{a^n b^n : n \geq 0\}$ Select

$L = \{a^n b^n : n \geq 0\}$ **Regular Pumping Lemma**

Objective: Prevent the computer from finding a valid partition.

Clear All | Explain | **My Attempts:**

1. I have selected a value for m, displayed below.
12

2. Please enter a possible value for w and press "Enter".
aaaaaaaaaaaabbbbbbbbbbbb

3. I have decomposed w into the following...
X = aaaaaaaaa; Y = aaa; Z = bbbbbbbbbbbb

4. Please enter a possible value for i and press "Enter".
i: 3 pumped string: aaaaaaaaaaaaaaaaaabbbbbbbbbbbb

5. Animation

```
           x        y          z
   w = aaaaaaaaa  aaa  bbbbbbbbbbbb
```

Selezionando invece "*You go first*", dove il computer dimostra che l'espressione data non è regolare fornendo una contraddizione come appena descritto:

$L = \{a^n b^n : n \geq 0\}$ **Regular Pumping Lemma**

Objective: Find a valid partition that can be pumped.

Clear All | Explain | **My Attempts:**

1. Please select a value for m in Box 1 and press "Enter".
7

2. I have selected w such that |w| >= m. It is displayed in Box 2.
aaaaaaabbbbbbb

3. Select decomposition of w into xyz.
x: aaa |x|: 3
y: aaaa |y|: 4 Set xyz
z: bbbbbbb |z|: 7
a a a a a a a b b b b b b b

4. I have selected i to give a contradition. It is displayed in Box 4.
i: 0 pumped string: aaabbbbbbb

5. Animation

```
        x    y      z
w = aaa aaaa bbbbbbb
```

<u>*Domande Wooclap*</u>

Quanti stati ci sono nell'automa che riconosce il linguaggio $\{0^n 1^n \mid n \geq 0\}$?

La rappresentazione condurrebbe ad un numero infinito di stati e non sarebbe più un automa a stati finiti.
Risposta corretta: Infiniti

Il linguaggio $\{0^n 1^n \mid n \geq 0\}$ è regolare?

Si intende per “regolare” un linguaggio riconoscibile da un automa a stati finiti che lo riconosce.
Questo non è il caso.
Risposta corretta: NO
Attenzione: l’elevazione a potenza ad “n” non è infatti permessa nei linguaggi regolari.

Per applicare il Pumping Lemma ad un linguaggio, consideriamo una stringa w che appartiene a L di lunghezza $\geq k$ e la spezziamo in _________ parti

Risposta: 3

Se selezioniamo una stringa w tale che $w \in L$ e $w = xyz$, quale delle seguenti parti non può essere una stringa vuota?

Risposta: *y*

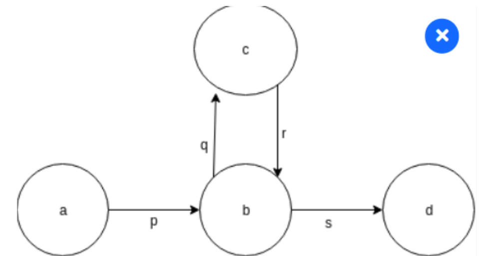
Dato un linguaggio L e una stringa w tale che $w \in L$, $w = xyz$ e $|w| \geq k$ per un numero intero costante k. Quale può essere la lunghezza massima della sottostringa xy, ossia $|xy| \leq ?$

Risposta: *k*

Rispondi in conformità con la terza e ultima affermazione del Pumping Lemma:
Per ogni _______ $xy^iz \in L$

Risposta: i >= 0

Se d è uno stato finale, quale delle seguenti affermazioni è corretta secondo il diagramma?



Risposta:

$x = p, y = qr, z = s$

Quali delle seguenti affermazioni sono vere? (puoi scegliere più di una risposta)

Risposte:
- Se un linguaggio è regolare, allora rispetta le condizioni del Pumping Lemma
- Se un linguaggio NON rispetta le condizioni del Pumping Lemma, allora NON è regolare

Possiamo usare il Pumping Lemma per (puoi scegliere più di una risposta):

Risposte:
Se dimostro L non rispetta P.L. (pumping lemma) → L non è regolare
Se dimostro che L rispetta P.L. → Non posso dire niente
Per dimostrare che è regolare → Trovo un automa o un'espressione regolare

Il linguaggio $\{a^nb^n : n \geq 0\}$ è regolare?

Risposta: NO

<u>Utile per esame:</u> *<u>Provare che un linguaggio sia regolare</u>*

Dal link https://cs.stackexchange.com/questions/1331/how-to-prove-a-language-is-regular

Another method, not covered by the answers above, is *finite automaton transformation*. As a simple example, let us show that the regular languages are closed under the *shuffle* operation, defined as follows:

$$L_1 \, S \, L_2 = \{x_1y_1 \dots x_ny_n \in \Sigma^* : x_1 \dots x_n \in L_1, y_1 \dots y_n \in L_2\}$$

You can show closure under shuffle using closure properties, but you can also show it directly using DFAs. Suppose that $A_i = \langle \Sigma, Q_i, F_i, \delta_i, q_{0i} \rangle$ is a DFA that accepts L_i (for $i = 1, 2$). We construct a new DFA $\langle \Sigma, Q, F, \delta, q_0 \rangle$ as follows:

- The set of states is $Q_1 \times Q_2 \times \{1, 2\}$, where the third component remembers whether the next symbol is an x_i (when 1) or a y_i (when 2).
- The initial state is $q_0 = \langle q_{01}, q_{02}, 1 \rangle$.
- The accepting states are $F = F_1 \times F_2 \times \{1\}$.
- The transition function is defined by $\delta(\langle q_1, q_2, 1 \rangle, \sigma) = \langle \delta_1(q_1, \sigma), q_2, 2 \rangle$ and $\delta(\langle q_1, q_2, 2 \rangle, \sigma) = \langle q_1, \delta_2(q_2, \sigma), 1 \rangle$.

Guessing often involves also verifying. One simple example is closure under *rotation*:

$$R(L) = \{yx \in \Sigma^* : xy \in L\}.$$

Suppose that L is accepted by the DFA $\langle \Sigma, Q, F, \delta, q_0 \rangle$. We construct an NFA $\langle \Sigma, Q', F', \delta', q'_0 \rangle$, which operates as follows. The NFA first guesses $q = \delta(q_0, x)$. It then verifies that $\delta(q, y) \in F$ and that $\delta(q_0, x) = q$, moving from y to x non-deterministically. This can be formalized as follows:

- The states are $Q' = \{q'_0\} \cup Q \times Q \times \{1, 2\}$. Apart from the initial state q'_0, the states are $\langle q, q_{curr}, s \rangle$, where q is the state that we guessed, q_{curr} is the current state, and s specifies whether we are at the y part of the input (when 1) or at the x part of the input (when 2).
- The final states are $F' = \{\langle q, q, 2 \rangle : q \in Q\}$: we accept when $\delta(q_0, x) = q$.
- The transitions $\delta'(q'_0, \epsilon) = \{\langle q, q, 1 \rangle : q \in Q\}$ implement guessing q.
- The transitions $\delta'(\langle q, q_{curr}, s \rangle, \sigma) = \langle q, \delta(q_{curr}, \sigma), s \rangle$ (for every $q, q_{curr} \in Q$ and $s \in \{1, 2\}$) simulate the original DFA.
- The transitions $\delta'(\langle q, q_f, 1 \rangle, \epsilon) = \langle q, q_0, 2 \rangle$, for every $q \in Q$ and $q_f \in F$, implement moving from the y part to the x part. This is only allowed if we've reached a final state on the y part.

Another variant of the technique incorporates bounded counters. As an example, let us consider change *edit distance closure*:

$$E_k(L) = \{x \in \Sigma^* : \text{ there exists } y \in L \text{ whose edit distance from } x \text{ is at most } k\}.$$

Given a DFA $\langle \Sigma, Q, F, \delta, q_0 \rangle$ for L, e construct an NFA $\langle \Sigma, Q', F', \delta', q'_0 \rangle$ for $E_k(L)$ as follows:

- The set of states is $Q' = Q \times \{0, \ldots, k\}$, where the second item counts the number of changes done so far.
- The initial state is $q'_0 = \langle q_0, 0 \rangle$.
- The accepting states are $F' = F \times \{0, \ldots, k\}$.
- For every q, σ, i we have transitions $\langle \delta(q, \sigma), i \rangle \in \delta'(\langle q, i \rangle, \sigma)$.
- Insertions are handled by transitions $\langle q, i + 1 \rangle \in \delta'(\langle q, i \rangle, \sigma)$ for all q, σ, i such that $i < k$.
- Deletions are handled by transitions $\langle \delta(q, \sigma), i + 1 \rangle \in \delta'(\langle q, i \rangle, \epsilon)$ for all q, σ, i such that $i < k$.
- Substitutions are similarly handles by transitions $\langle \delta(q, \sigma), i + 1 \rangle \in \delta'(\langle q, i \rangle, \tau)$ for all q, σ, τ, i such that $i < k$.

Pumping lemma: esercizi e dimostrazioni

All’interno del gioco, quindi, uno dei due giocatori possiede la corretta strategia vincente, facendo in modo che qualsiasi combinazione possa tentare l’avversario, abbia la vittoria.

Se il linguaggio rispetta il Pumping Lemma:

- il Giocatore 1 ha una strategia vincente
- per qualsiasi mossa faccia il Giocatore 2, può rispondere in modo da vincere il gioco.

Se il linguaggio non rispetta il Pumping Lemma:

- il Giocatore 2 ha una strategia vincente
- per qualsiasi mossa faccia il Giocatore 1, può rispondere in modo da vincere il gioco.

1 Sia L_{ab} il linguaggio delle stringhe sull'alfabeto $\{a, b\}$ dove il numero di a è uguale al numero di b. L_{ab} è regolare?

No, L_{ab} non è regolare:

- supponiamo per assurdo che lo sia
- sia k la lunghezza data dal Pumping Lemma
- consideriamo la parola $w = a^k b^k$
- sia $w = xyz$ una suddivisione di w tale che $y \neq \varepsilon$ e $|xy| \leq k$:
  $w = \underbrace{aaa\ldots a}_{x}\underbrace{a\ldots a}_{y}\underbrace{a\ldots ab\ldots bb}_{z}$
- poiché $|xy| \leq k$, le stringhe x e y sono fatte solo di a
- per il Pumping lemma, anche $xy^2z \in L_{ab}$, ma contiene più a che $b \Rightarrow$ assurdo

In questo esempio basterebbe anche qualsiasi valore diverso da 1 per la dimostrazione della non regolarità.

2 Il linguaggio $L_{rev} = \{ww^R : w \in \{a, b\}^*\}$ è regolare?

No, L_{rev} non è regolare:

- supponiamo per assurdo che lo sia
- sia k la lunghezza data dal Pumping Lemma
- consideriamo la parola $w = a^k bba^k$
- sia $w = xyz$ una suddivisione di w tale che $y \neq \varepsilon$ e $|xy| \leq k$:
  $w = \underbrace{aaa\ldots}_{x}\underbrace{\ldots a}_{y}\underbrace{abbaaa\ldots aaa}_{z}$
- poiché $|xy| \leq k$, le stringhe x e y sono fatte solo di a
- per il Pumping lemma, anche $xy^0z = xz \in L_{rev}$, ma non la posso spezzare in $ww^R \Rightarrow$ assurdo

Alcuni esempi di parole sono:

abaaaaba babbbbab aabbaa

(mancherebbero le palindrome di lunghezza dispari, ma sarebbero comprese le palindrome di lunghezza pari)

Il linguaggio non è regolare.

DIM: Per assurdo assumo che L_{rev} sia regolare

1) Sia "k" la lunghezza che mi fa rispettare il Pumping Lemma
2) Consideriamo la parola $w=a^k a^k$
3) Siano x, y, z tale che $w=xyz$, $y \neq \varepsilon$, $|xy|<=k$
   Considero l'esponente i=4
   $x=a^p$, $y=a^q$, $z=a^{k-p-q}a^k$
   $xy^3z= a^p a^{4q} a^{k-p-q}a^k = a^{k+3q}a^k = a^{2k+3q} \notin L_{rev}$

La parola qui scelta non andrebbe bene, perché non rispetta come al solito $|xy| \leq k$

3 Il linguaggio $L_{nm} = \{a^n b^m : n \text{ è dispari oppure } m \text{ è pari}\}$ è regolare?

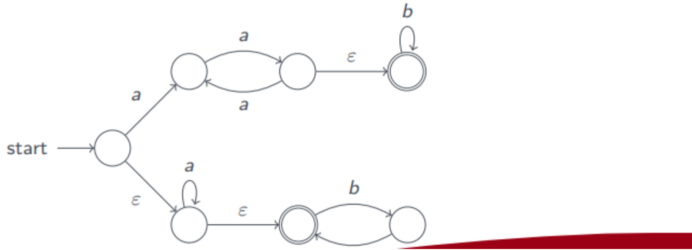
Dim: Suppongo per assurdo L_{nm} regolare.

1) Sia k la lunghezza del P.L.
2) Considero come parola $w= a^{2k+1}b^{2k}$
3) Siano x, y, z tale che $w=xyz$, $y \neq \varepsilon$, $|xy|<=k$
4) $x=a^p$, $y=a^q$, $z=a^{2k-1-p-q}b^{2k}$
   $i=2 xy^2z= a^{2k-1-p+q}b^{2k} \in L$

La parola sta invece nel linguaggio; infatti tutto si può costruire con un automa, come nel caso seguente:

Si, L_{nm} è regolare:

- è rappresentato dall'espressione regolare $\mathbf{a(aa)^*b^* + a^*(bb)^*}$
- e riconosciuto dall'automa



4 Il linguaggio $L_p = \{1^p : p \text{ è primo}\}$ è regolare?

Qui il ragionamento è interessante, dato che ragiona solo sugli esponenti, quindi:

supponiamo per assurdo che lo sia

sia k la lunghezza data dal Pumping Lemma

consideriamo una parola $w = 1^p$ con p primo e $p > k + 2$

sia $w = xyz$ una suddivsione di w tale che $y \neq \varepsilon$ e $|xy| \leq k$:

$$w = \underbrace{11\ldots11}_{x}\underbrace{11\ldots1}_{y}\underbrace{111\ldots11}_{z}$$

L'esercizio non è semplice; infatti si può ragionare ponendo come esponente il numero primo subito maggiore di quello attuale, per esempio "r" > p + 2, dove forziamo l'inserimento del numero del linguaggio. Mettiamo 2 perché abbiamo bisogno di almeno 2 numeri tali da ottenere la condizione di numero primo (il numero per sé stesso ed 1, nella pratica). Per esempio prendiamo o^r e ci interessa fondamentalmente solo la lunghezza, non la stringa per se (composta infatti di soli 0). Quindi prendiamo $xy^iz = xz + i|y|$ (perché dobbiamo buttarci dentro almeno un numero primo.

La stringa, abbiamo detto prima, deve avere almeno 2 caratteri oltre alla stringa stessa per essere "p" stessa. Ponendo i=|xz| nell'espressione precedente, otteniamo |xz|(1+|y|). Componendo in questo modo la stringa, riusciamo a dimostrare che la stringa non è prima, in quanto entrambi i fattori sono maggiori di 1.

Tecnicamente si vede da qui:

sia $|y| = m$: allora $|xz| = p - m$

per il Pumping lemma, anche $v = xy^{p-m}z \in L_p$

allora $|v| = m(p - m) + p - m = (p - m)(m + 1)$ si può scomporre in due fattori

poiché $y \neq \varepsilon$, allora $|y| = m > 0$ e $m + 1 > 1$

anche $p - m > 1$ perché abbiamo scelto $p > k + 2$ e $m \leq k$ perché $|xy| \leq k$

i due fattori sono entrambi maggiori di 1 e quindi $|v|$ non è un numero primo

$v \notin L_{rev}$, assurdo

<u>Domande Wooclap</u>

Ordina le mosse del gioco del Pumping Lemma

1) Giocatore 1 sceglie una lunghezza

2) Giocatore 2 sceglie la parola
3) Giocatore 1 spezza la parola
4) Giocatore 2 sceglie un esponente

Il linguaggio $L = \{a^\ell b^m c^n \mid \ell, m, n \geq 0$ e se $\ell = 1$ allora $m = n\}$ è regolare?

Risposta: SI

Dim: Supponiamo L sia non regolare
1) Sia k la lunghezza del P.L.
2) Consideriamo $w=a^l b^k c^k$ $\quad l+m < k$
3) Siano x, y, z tale che w=xyz, $y \neq \varepsilon$, $|xy|<=k$
4) Se y contiene almeno una b → i=z $xy^2z \notin L$

$x=\varepsilon$, $y=a$, $z=b^k c^k$
Quando l'esponente è diverso da 1, allora è regolare
Di fatto rispetta le condizione del PL, ma non è regolare
$xyz= a^i b^k c^k \in L$

Dim: L rispetta le condizioni del P.L.
1) La lunghezza k=1
2) sia $w=a^l b^m c^n \in L$
3) considero che $x=\varepsilon$, $y=a$, $\quad$ se $l > 0$ $z=a^{l-1} b^m c^n$

se l=0 $\quad$ y=b e m > 0
se m=0 $\quad$ y=c

4) $\forall$ i, $xy^iz \in L$
   se y=b → $xy^iz = b^i b^{m-1} c^n \in L$
   se $y=\varepsilon$ → $ac^l c^{n-1} b \in L$

   y=a se l=1 → $xy^iz= a^l b^m c^n \in L$
   se l diverso da 1
   $xy^l= a^i a^{l-1} b^n c^m \in L$

Grammatiche Context-Free

Considera il linguaggio

$$L_1 = \{a^\ell b^m c^n \mid \ell, m, n \geq 0 \text{ e se } \ell = 1 \text{ allora } m = n\}.$$

(a) Mostra che L_1 non è regolare.

(b) Mostra che L_1 si comporta come un linguaggio regolare rispetto al Pumping Lemma. In altre parole, dai una lunghezza del pumping k e dimostra che L_1 soddisfa le condizioni del Pumping Lemma per questo valore di k.

(c) Spiega perché i punti (a) e (b) non contraddicono il Pumping Lemma.

a) Per assurdo suppongo L_1 sia regolare. Quindi esiste un DFA A che riconosce L_1 A possiede K stati.
Su input ab^k

con uno stato $P_i = P_j$ per qualche $i \neq j$
Su input ab^ic^i $r_i \in F$

Su input ab^jc^i ma $ab^jc \notin L_1$ -> Contraddizione

A, B sono regolari -> A ∩ B è regolare
A ∩ B <u>non</u> è regolare → A, B non possono essere entrambi regolari
$L_1 \cap \{ab^*c^*\}=\{ab^nc^n \mid n \geq 0\}$ → Esercizio: usare il PL per dimostrare che non è regolare

P.L. Se L è regolare allora:
2) $\exists\, k \geq 0 \mid \forall\, w \in L, |w| \geq k$
$w=xyz$ t.c. $y \neq \varepsilon$ $|xy|<=k$ $\forall\, xy^iz \in L$ con $i >= 0$
Dim: che L_1 rispetta condizioni P.L.
Poniamo come K=2
Data una qualsiasi parola $w=a^lb^mc^n \in L$, con $|w| >= 2$, abbiamo 4 casi:
- $w = ab^mc^m$ $x = \varepsilon$, $y=a$, $z=b^mc^m$, $a^ib^ic^m \in L_1$, $\forall i >= 0$
- $w = aab^nc^n$ $x = \varepsilon$, $y=aa$, $z=b^mc^n$, $(aa)^ib^mc^n \in L_1$, $\forall i >= 0$
- $w = a^l\, b^m\, c^n$ con $l >= 3$ $x = \varepsilon$, $y=a$, $z=b^mc^m$, $a^{l-1}\, b^m\, c^n$ → $a^i\, a^{l-1}b^mc^n \in L_1\, \forall i >= 0$
- $w = b^mc^n$ $x = \varepsilon$ $y=$ primo carattere (b oppure c) $c=b^{n-1}c^n$ oppure $z=c^{n-1}$ → $xy^iz \in L_1\, \forall i >= 0$

Abbiamo visto linguaggi non regolari; abbiamo invece linguaggi liberi da contesto, chiamati <u>*context-free languages*</u>, vedendo due modi per descriverli:

- grammatiche context-free (CFG)
- automi a pila (pushdown automata)

Vediamo subito un esempio di grammatica context-free:

La grammatica G_1:

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

- insieme di regole di sostituzione (o produzioni)
- variabili: A, B
- terminali (simboli dell'alfabeto): $0, 1, \#$
- variabile iniziale: A

Una grammatica genera stringhe attraverso la *derivazione*:

- si scrive la variabile iniziale
- si trova una variabile che è stata scritta e una regola che inizia con quella variabile. Si sostituisce la variabile con il lato destro della regola
- si ripete (fino a quando non ci sono più variabili)

Esempio per G_1:

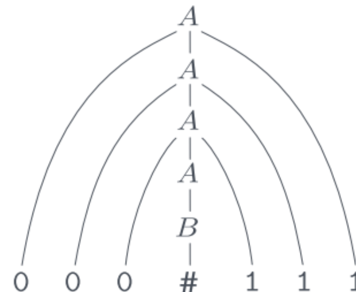
$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000\#111$$

La sequenza di sostituzioni si chiama derivazione di 000#111

In pratica si sostituisce tra 0A1 e 00A11 l'espressione "0A1" al posto di A
Poi arrivo ad A e la sostituirò con B
B sarà sostituita da # lasciando tre 1

A partire dalla parola, posso usare *l'albero sintattico* (*parse tree*), avendo come radice il simbolo iniziale e avendo come figli l'applicazione delle regole sulle variabili nella derivazione. Essi danno tutta la struttura della parola. Infatti:

Una derivazione definisce un albero sintattico (parse tree):



- la radice è la variabile iniziale
- i nodi interni sono variabili
- le foglie sono terminali

È possibile comporre attraverso le grammatiche delle frasi e delle parole simili al linguaggio naturale.
<SENTENCE> → <NOUN_PHRASE><VERB_PHRASE> (quindi si spezza la frase in sostantivo e verbo)
→ <CMPL_NOUN><VERB_PHRASE>
→ <CMPL_NOUN><CMPLX_VERB> → <ARTICLE><NOUN><CMPLX_VERB>
→ <ARTICLE><NOUN><VERB> → a<NOUN><VERB> → a <NOUN> touches → a boy touches
Con la grammatica riportata:

$$\begin{aligned}\langle SENTENCE\rangle &\Rightarrow \langle NOUN_PHRASE\rangle\langle VERB_PHRASE\rangle\\ &\Rightarrow \langle CMPLX_NOUN\rangle\langle VERB_PHRASE\rangle\\ &\Rightarrow \langle ARTICLE\rangle\langle NOUN\rangle\langle VERB_PHRASE\rangle\\ &\Rightarrow \texttt{a}\ \langle NOUN\rangle\langle VERB_PHRASE\rangle\\ &\Rightarrow \texttt{a boy}\ \langle VERB_PHRASE\rangle\\ &\Rightarrow \texttt{a boy}\ \langle CMPLX_VERB\rangle\\ &\Rightarrow \texttt{a boy}\ \langle VERB\rangle\\ &\Rightarrow \texttt{a boy sees}\end{aligned}$$

Tra le stringhe nel linguaggio di G_2 ci sono:

```
a boy sees
the boy sees a flower
a girl with a flower likes the boy
```

seguendo poi con esempi di derivazione:

Ecco quindi le regole formali di una grammatica context-free:

Definition

Una grammatica context-free è una quadrupla (V, Σ, R, S), dove
- V è un insieme finito di variabili
- Σ è un insieme finito di terminali disgiunto da V
- R è un insieme di regole, dove ogni regola è una variabile e una stringa di variabili e terminali
- $S \in V$ è la variabile iniziale

Le parole che stanno nel linguaggio sono quelle per cui possiamo trovare una derivazione:

Se u, v, w sono stringhe di variabili e terminali e $A \to w$ è una regola:
- uAv produce uwv: $uAv \Rightarrow uwv$
- u deriva v: $u \Rightarrow^* v$ se:
  - $u = v$, oppure
  - esiste una sequenza $u_1, u_2, \ldots, u_k$ tale che $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$
- il linguaggio della grammatica è $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

Linguaggi context-free

linguaggi generati da grammatiche context-free

u A v → u w v   w, u, v ∈ (V ∪ E)*   A -> W
u →* v   u = v
u → u_1 → u_2 → ..... → u_k → v
L(G) = {v ∈ Σ* | S →* v

__Attenzione__
L'idea intuitiva, per capire che il linguaggio è corretto, è di attuare delle derivazioni partendo dalla leftmost fino alle rightmost, a quel punto si verifica se la propria CFG è corretta.

__Domande Wooclap:__

Qual è il linguaggio generato dalla grammatica G_1?

Risposta:

$0^n\#1^n \mid n \geq 0$

Associa la notazione con la definizione:

$A \to uAv$

B. Regola

$uAv \Rightarrow uwv$

A. Produzione

$S \Rightarrow^* u$

C. Derivazione

Descrivi il linguaggio generato dalla grammatica $S \to (S) \mid SS \mid \varepsilon$

Si intende:
S → (S)
S → SS
S → ε
La regola dice che per ogni parentesi aperta ci sta una parentesi chiusa:
S → (S) → ((S))
→ (((S))) → ((()))
S → SS → (S)S → (SS)S → ((S)S)S) → (()S)S → (()(S))S →
(()())S → (()())(S) → (()())((S)) → (()())(())

Descrivi il linguaggio generato dalla grammatica

$$\langle EXPR\rangle \to \langle EXPR\rangle + \langle TERM\rangle \mid \langle TERM\rangle$$
$$\langle TERM\rangle \to \langle TERM\rangle \times \langle FACTOR\rangle \mid \langle FACTOR\rangle$$
$$\langle FACTOR\rangle \to (\langle EXPR\rangle) \mid a$$

<EXPR> → <EXPR> + <TERM>
<EXPR> → <TERM>
<TERM> → <TERM> X <FACTOR>
<TERM> → <FACTOR>
<FACTOR> → (<EXPR>)
<FACTOR>→ a
Risposta: Insieme di somme e moltiplicazioni
a + a x a
(a + a) x a
(a + a + a) x (a x a) + (a+a)n

Progettare grammatiche context-free

La progettazione di una grammatica è un processo creativo, non meccanico; esistono tuttavia delle tecniche utili che si possono usare per questo scopo. Molti linguaggi sono *unione di linguaggi più semplici*, costruendo grammatiche per ogni componente e poi unendo le grammatiche con nuova regola iniziale.

L={0^n1^n | n >= 0} U {1^n0^n | n >=0}
S_1 → $0S_11$ | ε

$S_2 \rightarrow 1S_20 \mid \varepsilon$
$S \rightarrow S_1 \mid S_2$ (dove "S" sta per variabile iniziale)

Suddividiamo quindi in linguaggio per vari casi e progettiamo la grammatica per ognuno di questi.

Esempio: grammatica per $\{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$

- Grammatica per 0^n1^n: $S_1 \rightarrow 0S_11 \mid \varepsilon$
- Grammatica per 1^n0^n: $S_2 \rightarrow 1S_20 \mid \varepsilon$
- Unione: $S \rightarrow S_1 \mid S_2$

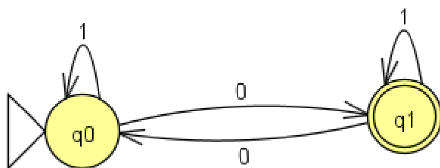
Se il linguaggio è regolare, possiamo trovare un DFA che lo riconosce.
L'idea è quindi di trasformare il DFA in grammatica:

- una variabile R_i per ogni stato q_i
- una regola $R_i \rightarrow aR_j$ per ogni transizione $\delta(q_i, a) = q_j$
- una regola $R_i \rightarrow \varepsilon$ per ogni stato finale q_i
- R_0 variabile iniziale, se q_0 è lo stato iniziale

Vediamo quindi:

Esempio

$\{w \in \{0,1\}^* \mid w$ contiene un numero dispari di $0\}$

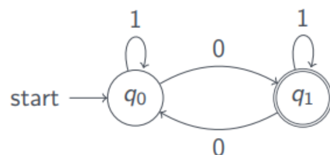


$V = \{R_0, R_1\}$
$\Sigma = \{0,1\}$
$R_0 \rightarrow 0R_1 \mid 1R_0$
$R_1 \rightarrow 0R_0 \mid 1R_1 \mid \varepsilon$

Applico le varie produzioni secondo le regole e:
$R_0 \rightarrow 0R_1 \mid 00R_0 \rightarrow 001R_0 \rightarrow 0010R_1 \rightarrow 0010$

L'idea è che ogni linguaggio regolare è anche un linguaggio context-free.
Alla fine:

- DFA per $\{w \in \{0,1\}^* \mid w$ contiene un numero dispari di $0\}$:



- Grammatica context-free:

$$R_0 \rightarrow 0R_1 \mid 1R_0$$
$$R_1 \rightarrow 0R_0 \mid 1R_1 \mid \varepsilon$$

Ad esempio: riconoscere $\{0^n1^n \mid n >=0\}$ richiede di contare gli 0 e il loro numero deve essere uguale agli 1.
L'idea è:
- avere regole nella forma R → uRv generando stringhe dove "u" corrisponde a "v"
- ad esempio la grammatica S = 0S1 | ε dice letteralmente "ripetizione ricorsiva della stringa S con num. di 0 uguale al numero di 1

Quindi basta avere una variabile che si ripete in mezzo a due altre cose e strutture, riconoscendo il linguaggio. Le stringhe possono quindi contenere strutture che appaiono in modo ricorsivo.
L'idea è quindi di porre nelle regole la variabile che genera la struttura nei punti in cui questa può apparire ricorsivamente.
Esempio delle operazioni aritmetiche (sostituzione di a con un'intera espressione aritmetica parentesizzata, o anche la regola <FACTOR> → (<EXPR>) | a ci dice che possiamo effettuare questa sostituzione.
Qui abbiamo: somma, moltiplicazione, fattorizzazione con parentesi come descritto.
<E> → <E> + <T> | <T>
<T> → <T> X <F> | <F>
<F> a | (<E>)

Quindi potremmo sostituire "a" con qualsiasi espressione rimpiazzandola ricorsivamente.

a + a	a x a
(a x a) + a	(a + a) x a) + a

Quali di queste regole fanno parte della grammatica per $\{a^ib^jc^k \mid i = j$ oppure $j = k\}$?

T → aTb|ε	a^nb^n
U → cU | ε	c^k
S → VZ	
V → aV|ε	
Z → bZc|ε	

Come potete creare la grammatica per $\{w \mid w$ contiene un numero pari di 0 e dispari di $1\}$?

Risposta: Convertendo il DFA per il linguaggio in grammatica

Quali di queste regole fanno parte della grammatica per le espressioni regolari?

Nota: sono tutte corrette

$EXPR \to EXPR + EXPR \mid TERM$ ✓

$TERM \to TERM \cdot TERM \mid POWER$ ✓

$POWER \to FACTOR* \mid FACTOR$

$FACTOR \to (EXPR) \mid 0 \mid 1 \mid \emptyset \mid \varepsilon$ ✓

(0 + 1)*
E → T → P → F* → (E)* → (E+E)* →> (T+E)* → (P+E)* → (F+E)* → (0+E)* → (0+T)* → (0+F)* → (0+1)*

Descrivi (in italiano) due significati diversi per la frase "the girl touches the boy with the flower"
Inserisci una risposta diversa per ognuno dei due significati

La frase è appositamente ambigua, infatti la grammatica può generare una parola ambiguamente, avendo alberi sintattici diversi che permettono di ottenere quella stringa.
Infatti la grammatica che useremmo sarebbe quella vista nella scorsa lezione (sentence, noun_phrase, verb_phrase, ecc). Si potrebbe quindi avere interpretazione diversa per la stessa stringa.

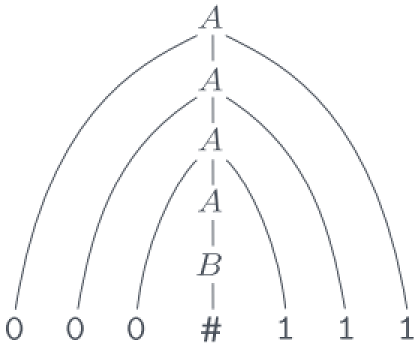
Data una grammatica $G = (V, \Sigma, R, S)$, un albero sintattico è un albero che soddisfa le seguenti condizioni:

1. i nodi interni sono variabili di V
2. le foglie sono, variabili, simboli terminali o ε
3. Se un nodo interno è etichettato con A e i suoi figli sono, da sinistra a destra
$$X_1, X_2, \dots, X_k$$
allora $A \to X_1 X_2 \dots X_k$ è una regola di G

Possiamo generare gli alberi sintattici per le stringhe che stanno nel linguaggio di G nel seguente modo:

1. Usa la variabile iniziale come radice dell'albero
2. Trova una foglia etichettata con una variabile A e una regola $A \to X_1 X_2 \dots X_k$. Aggiungi alla foglia i figli $X_1, \dots, X_k$ da sinistra a destra
3. Ripeti 2 fino a quando tutte le foglie sono etichettate con terminali o ε

generando il seguente albero sintattico (parse tree) per la grammatica G_1, dando quindi informazioni in merito alla sintassi e alla generazione di ogni simbolo della parola:

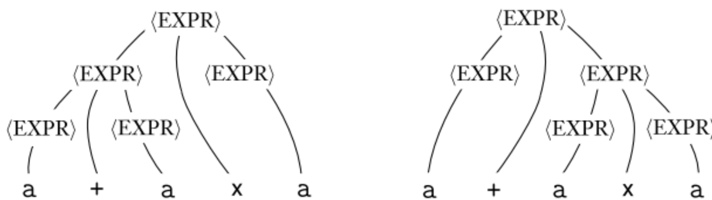


L'esempio di albero sintattico infatti precedente può essere generato in due modi diversi (infatti potrebbe non avere una precisa preferenza di esecuzione delle operazioni, ragion per cui nasce l'ambiguità).

G_5

$\langle EXPR \rangle \to \langle EXPR \rangle + \langle EXPR \rangle \mid \langle EXPR \rangle \times \langle EXPR \rangle \mid (\langle EXPR \rangle) \mid a$

Questa grammatica genera la stringa $a + a \times a$ in due modi diversi!



- Una grammatica genera ambiguamente una stringa se esistono due alberi sintattici diversi per quella stringa
- Attenzione! Derivazioni diverse possono portare allo stesso albero sintattico!
- Definiamo un ordine per le derivazioni:
  - Derivazione a sinistra (leftmost derivation): ad ogni passo, sostituisco la variabile che si trova più a sinistra.

In pratica quindi una stringa si genera ambiguamente a causa delle possibili doppie derivazioni ed una grammatica si considera ambigua se contiene almeno una stringa generata ambiguamente.
Esistono anche linguaggi context-free generati solamente da grammatiche ambigue sono *linguaggi inerentemente ambigui.*

- Esempio: il linguaggio $\{a^i b^j c^k \mid i = j \text{ oppure } j = k\}$

Esercizio

Dimostrare che $\{a^i b^j c^k \mid i = j \text{ oppure } j = k\}$ è inerentemente ambiguo.

La grammatica G:

$$S \to SS \mid T$$
$$T \to aTb \mid ab$$

è ambigua?

Risposta:
S → SS → SSS → TSS → abSS → abTS→ ababS → ababT → ababab

Per essere ambiguo:
- avere alberi sintattici diversi
- usare la tecnica della *leftmost derivation* e concludere che si arriva allo stesso linguaggio pur seguendo sostituzioni diverse

Infatti:
S → SS → TS → abS → abSS → abTS →ababS → ababab

Quindi: *si, è ambiguo*, come spiegato.

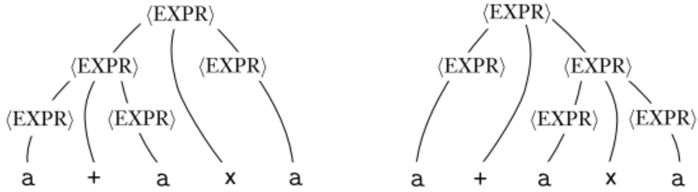
Grammatiche context-free/Forme Normali

Andiamo più nel dettaglio partendo dalla grammatica seguente, che ha un problema, avendo infatti le stringhe che portano a due derivazioni diverse (ambiguità):

G_5

$$\langle EXPR \rangle \to \langle EXPR \rangle + \langle EXPR \rangle \mid \langle EXPR \rangle \times \langle EXPR \rangle \mid (\langle EXPR \rangle) \mid a$$

partendo dall'albero sintattico che considera prima il prodotto poi la somma e viceversa, da un punto di vista operazionale:



partendo dalle espressioni:
(a + a) x a e a + (a x a)
calcolando quindi un valore sbagliato ed essere considerate ***ambigue***, generabile quindi attraverso molteplici alberi sintattici, non avendo regole sull'ordine di derivazione (si può arrivare allo stesso risultato, *cambiando l'ordine dei fattori il risultato non cambia*).

Il primo caso considerabile ad esempio è la derivazione a sinistra, risultando una grammatica ambigua se risulta una stringa generata almeno ambiguamente.
Una stringa è *equivalente* se dalla grammatica G esistono più derivazioni a sinistra che la generano.
In alcuni casi esistono dei linguaggi context-free ma generati solo da grammatiche ambigue, quindi *linguaggi inerentemente ambigui*, ad esempio:

il linguaggio $\{a^i b^j c^k \mid i = j \text{ oppure } j = k\}$
Nota: questo linguaggio rimane sempre ambiguo.

S → UT | VZ
U → aUb | ε $a^n b^n$
T → cT | ε c^n
V → aV | ε a^m
Z → bZc | ε $b^n c^n$

Diamo quindi l'esempio della derivazione con $a^n b^n c^n$

con esempio di stringa: *aabbcc*
ed esempio di derivazione a sinistra:
S → UT → aUbT → aaUbbT → aabbT →aabbcT →aabbccT → aabbcc

per poi derivare facendo un altro esempio:
S → VZ → aVz → aaVZ → aaZ → aabZc → aabbZcc → aabbcc

Parliamo poi della *forma normale di Chomsky*, considerata forma semplice ma potente.
Nota importante: questa viene utilizzata in tutti gli esercizi che dicono (*se il linguaggio X è context free allora Y è context free*). Bisogna infatti trasformare il linguaggio in questa forma e allora è considerabile normalmente context free. Segue quindi la definizione:

Definition

Una grammatica context-free è in forma normale di Chomsky se ogni regola è della forma

$$A \to BC$$
$$A \to a$$

dove a è un terminale, B, C non possono essere la variabile iniziale. Inoltre, ci può essere la regola $S \to \varepsilon$ per la variabile iniziale S

dove "a" è l'esempio di uno simbolo generico, con la regola che sia effettivamente unico, mentre per la prima regola è obbligatoriamente formato da due stringhe.

Possiamo dunque cercare di dimostrare il teorema:

Ogni linguaggio context-free è generato da una grammatica in forma normale di Chomsky

Idea: possiamo trasformare una grammatica G in forma normale di Chomsky:

1. aggiungiamo una nuova variabile iniziale
2. eliminiamo le ε-regole $A \to \varepsilon$
3. eliminiamo le regole unitarie $A \to B$
4. trasformiamo le regole rimaste nella forma corretta

Vediamo quindi un esempio:

Trasformiamo la grammatica G_6 in forma normale di Chomsky:

$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \varepsilon$$

In questo caso si vede che ci sono regole unitarie, ε-simboli e altro.

Per cominciare a trasformarla consideriamo (il testo barrato è quello eliminato):

1 aggiungiamo una nuova variabile iniziale $S_0 \notin V$ e la regola

$$S_0 \to S$$

In questo modo garantiamo che la variabile iniziale non compare mai sul lato destro di una regola

G'=(V', Σ , R', S_0)
dove si nota che S appare a destra e si introduce un nuovo stato iniziale.
S' → S
S → ASA |aB
A → B|S|ε
B→ b|ε
dove metto ε anche in A perché la regola successiva va in B che va a sua volta in ε.
Successivamente:

2 Eliminiamo le ε-regole $A \to \varepsilon$:

- se $A \to \varepsilon$ è una regola dove A non è la variabile iniziale
- per ogni regola del tipo $R \to uAv$, aggiungiamo la regola

$$R \to uv$$

- attenzione: nel caso di più occorrenze di A, consideriamo tutti i casi: per le regole come $R \to uAvAw$, aggiungiamo

$$R \to uvAw \mid uAvw \mid uvw$$

- nel caso di regole $R \to A$ aggiungiamo $R \to \varepsilon$ solo se non abbiamo già eliminato $R \to \varepsilon$
- Ripeti finché non hai eliminato tutte le ε-regole

rimuovo le ε-regole, quindi B → ε ed A → ε:
Rimuovendo B → ε (quindi vuol dire che considero tutte le stringhe dove B è nullo)

S' → S
S → ASA|aB|a
A → B|S|ε
B→b

e poi rimuovo A → ε (tutti i casi con ASA dove a è nullo e tolgo la ε):

S' → S
S → ASA|aB|a|AS|SA|S
A →B|S
B→b

applicando poi la terza parte:

3 Eliminiamo le regole unitarie $A \to B$:

- se $A \to B$ è una regola unitaria
- per ogni regola del tipo $B \to u$, aggiungiamo la regola

$$A \to u$$

a meno che $A \to u$ non sia una regola unitaria eliminata in precedenza
- Ripeti finché non hai eliminato tutte le regole unitarie

avendo come regole unitarie (regola che va verso un'altra regola non terminale), quindi:
S→S   S'→S   A→B   A→S

Partiamo rimuovendo S → S, che non fa nulla in pratica:
S' → S
S → ASA|aB|a|AS|SA
A →B|S
B→b

quindi eliminiamo S' → S (quindi sostituisco S con ASA|aB|a|AS|SA)
S' → ASA|aB|a|AS|SA
S → ASA|aB|a|AS|SA
A →B|S
B→b

poi eliminiamo A → B (quindi sostituisco B con b):
S' → ASA|aB|a|AS|SA
S → ASA|aB|a|AS|SA
A →b|S
B→b

ed infine eliminiamo A → S (quindi sostituisco S con ASA|aB|a|AS|SA):
S' → ASA|aB|a|AS|SA
S → ASA|aB|a|AS|SA
A →b| ASA|aB|a|AS|SA
B→b

4 Trasformiamo le regole rimaste nella forma corretta:

- se $A \to u_1 u_2 \dots u_k$ è una regola tale che:
  - ogni u_i è una variabile o un terminale
  - $k \geq 3$
- sostituisci la regola con la catena di regole

$$A \to u_1 A_1, \quad A_1 \to u_2 A_2, \quad A_2 \to u_3 A_3, \quad \dots \quad A_{k-2} \to u_{k-1} u_k$$

- rimpiazza ogni terminale u_i sul lato destro di una regola con una nuova variabile U_i, e aggiungi la regola

$$U_i \to u_i$$

- ripeti per ogni regola non corretta

Ora dobbiamo trovare le produzioni che hanno più di 2 variabili a destra:
S' → ASA   S → ASA   A → ASA

In pratica, vedendo che tutte hanno AS oppure SA come variabile rimpiazzabile, posso creare una nuova regola che le sostituisce, ottenendo:

S' → AX|aB|a|AS|SA
S → AX|aB|a|AS|SA
A → b|AX|aB|a|AS|SA
B→ b
X → SA

Adesso "a" è stato terminale e quindi dobbiamo cambiare tutte le produzioni che contengono "a" con una nuova regola (le espressioni sono S' → aB, S → ab, A → aB), aggiungendo come regola y → a:

S' → AX|YB|a|AS|SA
S → AX|YB|a|AS|SA
A → b|AX|YB|a|AS|SA
B→ b
X → SA
Y → a

Quindi in generale:

1)

If the Start Symbol S occurs on some right side, create a new Start Symbol S' and a new Production S' → S.

2)

Removal of Null Productions

In a CFG, a Non-Terminal Symbol 'A' is a nullable variable if there is a production A → ∈ or there is a derivation that starts at 'A' and leads to ∈. (Like A→................→∈)

Procedure for Removal:

Step 1: To remove A→∈, look for all productions whose right side contains A
Step2: Replace each occurences of 'A' in each of these productions with ∈
Step 3: Add the resultant productions to the Grammar

3)

Any Production Rule of the form A→B where A, B ∈ Non Terminals is called Unit Production

Procedure for Removal

Step 1: To remove A→B, add production A→x to the grammar rule whenever B→x occurs in the grammar. [x ∈ Terminal, x can be Null]
Step 2: Delete A→B from the grammar.
Step 3: Repeat from Step 1 until all Unit Productions are removed.

4)

Replace each Production $A \to B_1 \ldots\ldots B_n$ where n > 2, with $A \to B_1 C$ where $C \to B_2 \ldots\ldots B_n$
Repeat this step for all Productions having two or more Symbols on the right side.

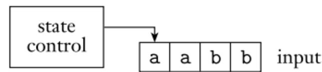
5)

If the right side of any Production is in the form A→aB where 'a' is a terminal and A and B are non-terminals, then the Production is replaced by A→XB and X→a.
Repeat this step for every Production which is of the form A→aB

Automi a pila/pushdown automata/PDA

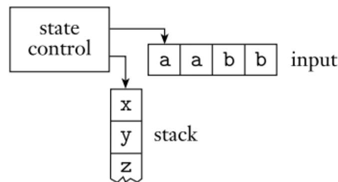
Finora tutti gli automi visti avevano queste caratteristiche, essendo a stati finiti:

- Input: stringa di caratteri dell'alfabeto
- Memoria: stati
- Funzione di transizione: dato lo stato corrente ed un simbolo di input, stabilisce quali possono essere gli stati successivi



passando poi a quelli a pila:

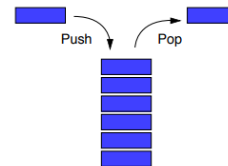
- Input: stringa di caratteri dell'alfabeto
- Memoria: stati + pila
- Funzione di transizione: dato lo stato corrente, un simbolo di input ed il simbolo in cima alla pila, stabilisce quali possono essere gli stati successivi e i simboli da scrivere sulla pila



L'automa dispone di una quantità di memoria potenzialmente infinita, con accesso limitato al suo contenuto, eseguito tramite operazioni di *push* (metti nella coda nell'ultima posizione) e *pop* (metti in prima posizione):

La pila è un dispositivo di memoria last in, first out (LIFO):

- Push: scrivi un nuovo simbolo in cima alla pila e "spingi giù" gli altri
- Pop: leggi e rimuovi il simbolo in cima alla pila (top)



La pila permette di avere memoria infinita (ad accesso limitato)

Un PDA usa la pila per contare 0 e 1:

- legge i simboli in input, e scrive ogni 0 letto sul
- non appena vede gli 1, cancella uno 0 dalla pila
- se l'input termina esattamente quando la pila si
- se ci sono ancora 0 nella pila al termine dell'inp
- se la pila si svuota prima della fine dell'input, ri
- se qualche 0 compare nell'input dopo gli 1, rifiu

Una parola che dovrebbe essere accettata (perché fa parte del linguaggio): 000111

Parole che vengono rifiutate: 001 0110 0100

Vediamo la definizione formale di ***automa a pila/pushdown automata:***

Un automa a pila (o Pushdown Automata, PDA) è una sestupla
$P = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

- Q è l'insieme finito di stati
- Σ è l'alfabeto di input
- Γ è l'alfabeto della pila
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \mapsto 2^{Q \times \Gamma_\varepsilon}$ è la funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme di stati accettanti

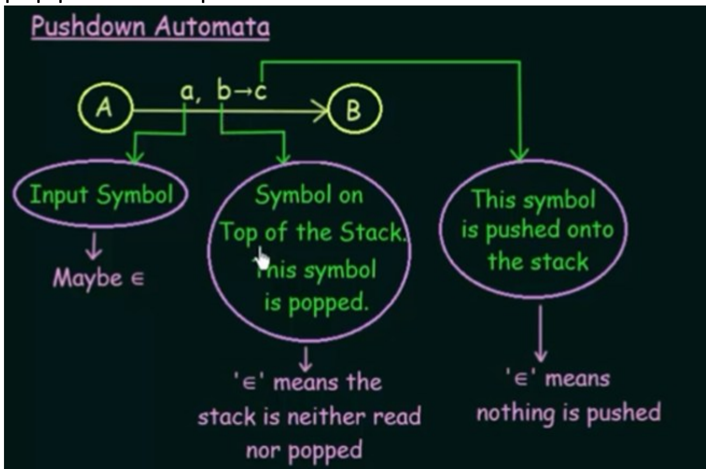
(dove $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$)

δ takes as argument a triple δ (q, a, X) where:
(i) q is a State in Q
(ii) a is either an Input Symbol in Σ or a = ϵ
(iii) X is a Stack Symbol, that is a member of Γ

The output of δ is finite set of pairs (p, γ) where:
p is a new state
γ is a string of stack symbols that replaces X at the top of the stack

Eg. If γ = ϵ then the stack is popped
If γ = X then the stack is unchanged
If γ = YZ then X is replaced by Z and Y is pushed onto the stack

La sostanza è la seguente: la pila legge un simbolo (input symbol), toglie un simbolo (pop), tale che il successivo simbolo immesso sulla cima della pila segua l'ordine inverso (LIFO), quindi eseguendo sempre un pop prima di un push.



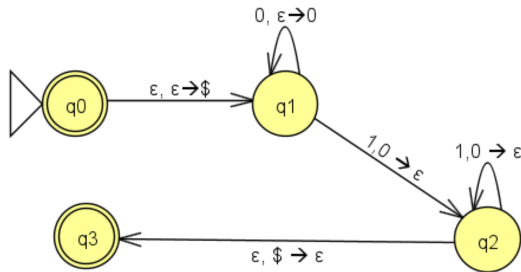
Vediamo poi un esempio:
$Q=\{q_0, q_1, q_2, q_3\}$ $\Sigma=\{0,1\}$
$\Gamma=\{0, \$\}$

\$ è simbolo speciale che serve per capire che la pila è vuota (in alcuni casi si usa Z_0) e questo rappresenta l'ultimo simbolo che dovrà essere rimosso dallo stack per essere considerabile corretto.

Descriviamo quindi la relativa tabella di transizione:
con δ descritta dalla tabella:

Input:	0			1			ε		
Pila:	0	\$	ε	0	\$	ε	0	\$	ε
q_0									$\{(q_1, \$)\}$
q_1			$\{(q_1, 0)\}$	$\{(q_2, \varepsilon)\}$					
q_2				$\{(q_2, \varepsilon)\}$				$\{(q_3, \varepsilon)\}$	
q_3									

e poi in forma di diagramma di transizione, convertito ad automa. Qui non si fa altro che immettere 011 come simboli di input, facendo push nell'ordine di \$, 0 e poi pop di 0, 0, \$:



La sostanza è descritta sopra, poi segue tutta la tabella. Si consideri che uno stack accetta una stringa che si esegue pop sullo stato iniziale e se lo stato finale viene raggiunto. Inoltre viene considerato q_0 stato finale perché il linguaggio potrebbe dover accettare *n=0* sull'alfabeto precedente.

Si vede seguendo le regole sopra che \$ viene buttato nello stack. Per le stesse regole di prima, non si poppa nulla dallo stack ma si pushano i due zeri consecutivamente. Successivamente non dobbiamo più pushare nulla, occorre solo fare il pop dei due zeri e successivamente dello stato iniziale \$.
Ecco quindi in maniera svolta tabellare quello che fa lo stack nel modo appena descritto (con una parola accettata, ad esempio 0011):

Stato	Pila
q_0	ε
q_1	\$
q_1	0\$
q_1	00\$
q_2	0\$
q_2	\$
q_3	ε

Poi diamo l'esempio della parola 011 (parola rifiutata):

Stato	Pila
q_0	ε
q_1	\$
q_1	0\$
q_2	\$
q_3	ε

e successivamente la parola 001 (sempre parola rifiutata):

Stato	Pila
q_0	ε
q_1	\$
q_1	0\$
q_1	00\$
q_2	0\$

Altro esempio con la parola w=ε0011ε (al di là di ε, la stringa viene accettata come si vede):
$(r_1, \$) \in (r_0, \varepsilon, a=\varepsilon)$

Stato	Pila	
$r_0=q_0$	ε	$=s_0=at=\varepsilon*e$
$r_1=q_1$	$	$=s_1=bt=\$*\varepsilon$
$r_2=q_1$	0$	$=s_2$
r_3q_1	00$	$=s_3$
$r_4=q_2$	0$	$=s_4$
$r_5=q_2$	$	$=s_5$
$r_6=q_3$	ε	$=s_6$

Un PDA accetta un linguaggio se:

Data una parola w, un PDA accetta la parola se:

- possiamo scrivere $w = w_1 w_2 \dots w_m$ dove $w_i \in \Sigma \cup \{\varepsilon\}$
- esistono una sequenza di stati $r_0, r_1, \dots, r_m \in Q$ e
- una sequenza di stringhe $s_0, s_1, s_2, \dots, s_m \in \Gamma^*$

tali che

1. $r_0 = q_0$ e $s_0 = \varepsilon$ (inizia dallo stato iniziale e pila vuota)
2. per ogni $i = 0, \dots, m-1$, $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ con $s_i = at$ e $s_{i+1} = bt$ per qualche $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$ (rispetta la funzione di transizione)
3. $r_m \in F$ (la computazione termina in uno stato finale)

oppure più in generale si arriva alla rimozione dello stato iniziale (si svuota tutta la pila o consuma tutto l'input):

- la nostra definizione di PDA accetta le parole per stato finale
- esiste un altro modo per definire la condizione di accettazione:

Accettazione per pila vuota

Un PDA accetta la parola w per pila vuota se esiste una computazione che

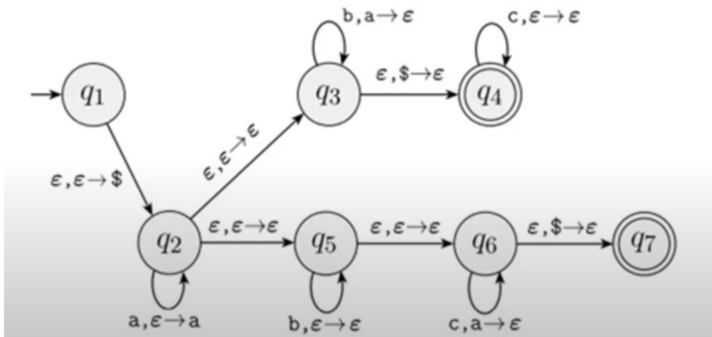
- consuma tutto l'input
- termina con la pila vuota ($s_m = \varepsilon$)

Equivalenza

Per ogni linguaggio accettato da un PDA per stato finale esiste un PDA che accetta per pila vuota, e viceversa.

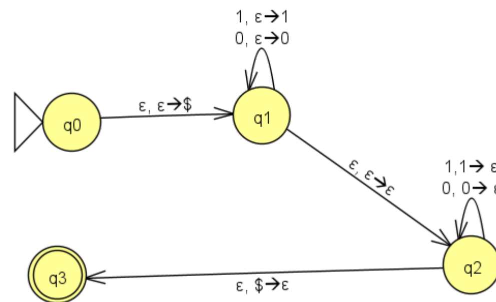
Vediamo gli esempi pratici:

1. Costruisci un PDA per il linguaggio $\{a^i b^j c^k \mid i = j$ oppure $j = k\}$



2. Costruisci un PDA per il linguaggio $\{ww^R \mid w \in \{0,1\}^*\}$, dove w^R indica la parola w scritta al contrario

L'osservazione è che, essendo la stringa palindroma, il primo passaggio butta lo stato iniziale che sarà l'ultimo ad essere rimosso, lo stato successivo andrà in se stesso con entrambi 0/1 (permesso perché è un automa non deterministico). A questo punto l'automa controlla idealmente se quello che c'era nello stack è uguale al resto della stringa, poppando a/b se sono nella cima dello stack e poi togliamo lo stato iniziale, accettando la stringa.
Un esempio di stringa accettata: *abba*, in quanto subito *b* è cima dello stack e viene tolto, assieme ad *a*.
Per lo stesso motivo, non viene accettata *abab*, in quanto non si rispetta l'ordine di pop.
Segue il PDA qui a lato:



<u>Domande Wooclap:</u>

Quali parole associ al termine "pila"?

Attenzione, la votazione è chiusa

Stack

Una pila è una memoria di tipo:

La risposta corretta era

LIFO (Last In, First Out)

FILO (First In, Last Out)

Quali tipi di simboli si possono trovare nella pila di un PDA?
1) Terminali 2) Variabili 3) Sia (1) che (2)
La risposta giusta è:
Nessuna delle precedenti

L'automa a pila si distingue per: *la presenza di una pila*

L'INPUT della funzione di transizione è costituito da (puoi scegliere più di una risposta):

lo stato attuale

il simbolo di input

il simbolo in cima alla pila

L'OUTPUT della funzione di transizione è:
un'insieme di coppie stato/simbolo della pila

Da CFG a PDA/Da PDA a CFG

Dal punto di vista dei linguaggi rappresentabili si ha l'uguaglianza tra grammatiche CF e PDA.

Un linguaggio è context-free se solo se esiste un PDA che lo riconosce

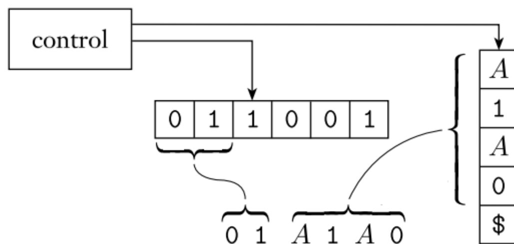
- Sappiamo che un linguaggio è context-free se esiste una CFG che lo genera
- Mostreremo come trasformare la grammatica in un PDA
- E viceversa, come trasformare un PDA in una grammatica

Partiamo dal dimostrare la *conversione da CFG a PDA* (partendo dal teorema di prima):

Idea.

- Se L è context free, allora esiste una CFG G che lo genera
- Mostriamo come trasformare G in un PDA equivalente P
- P è fatto in modo da simulare i passi di derivazione di G
- P accetta w se esiste una derivazione di w in G

Si parte quindi da una stringa, scegliendo variabili, regole da applicare e sostituiamo nella stringa in costruzione, fintanto che rimangono variabili.
Similmente, ogni derivazione si compone di stringhe intermedie memorizzabili da una pila P, la quale trova le variabili nella stringa intermedia e fa le sostituzioni seguendo la regole della grammatica G.
Idea: metto nella pila solo i simboli dalla prima variabile in poi

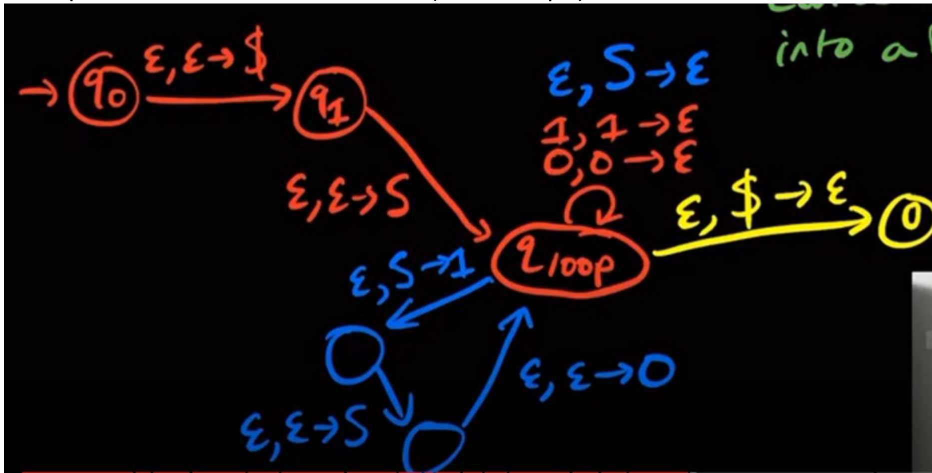


Considerando il simbolo iniziale $ e la variabile iniziale S sulla pila, l'automa sceglie non deterministicamente una regola, capendo se il simbolo in cima è variabile o terminale, rispetto ai prossimi simboli di input.

1. Inserisci il simbolo marcatore $ e la variabile iniziale S sulla pila
2. Ripeti i seguenti passi:
   1. Se la cima della pila è la variabile A: scegli una regola $A \to u$ e scrivi u sulla pila
   2. Se la cima della pila è un terminale a: leggi il prossimo simbolo di input.
      - se sono uguali, procedi
      - se sono diversi, rifiuta
   3. Se la cima della pila è $: vai nello stato accettante

Quindi vuol dire che se vedo dei terminali creo un loop che ha il simbolo stesso come input ed una serie di transizioni uscenti che ritornano nel loop per rappresentare le altre regole.
Per esempio, nel caso seguente completo:
- prima si fa il push di $ e di S
- successivamente si ha il loop dei simboli di input nell'ordine inverso (prima 0, essendo LIFO abbiamo che lo 0 andrà inserito per ultimo), questo nel caso in cui non ci sia nessuna stringa
- si creano poi transizioni che ritornano sullo stato corrente non annullando il lavoro fatto (per esempio, nel caso dell'1, semplicemente aggiungiamo uno stato esterno a cui segue una transizione in push di 1, un push del simbolo iniziale (quindi 0S1).

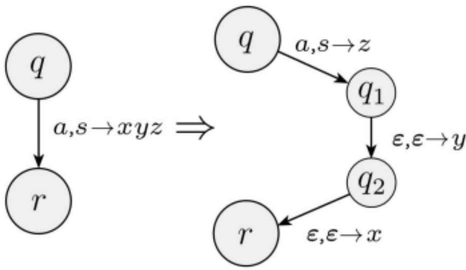
Esempio di PDA costruito dalla CFG (S → 0S1 | ε):



Supponiamo quindi che nella costruzione del PDA si vada da uno stato "q" ad uno stato "r", leggendo i simboli uno alla volta, inserendo la stringa u = xyz
L'accesso è sempre dall'alto e vogliamo ottenere:

x
y
z
$

Nell'automa che costruiamo, avremo una struttura di questo tipo:



avendo una sequenza di transizioni che descrivere raggruppando l'insieme di stati di un automa.
Formalmente:

Data $G = (V, \Sigma, R, S)$, definiamo $P = (Q, \Sigma, \Gamma, q_{start}, F)$:

- $Q = \{q_{start}, q_{loop}, q_{end}\}$
- $\Gamma = \Sigma \cup V \cup \{\$\}$
- $F = \{q_{end}\}$
- funzione di transizione:

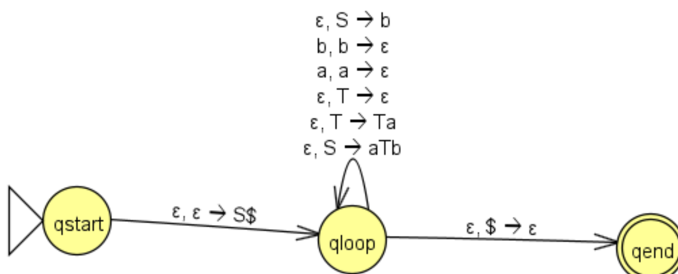
$\varepsilon, A \to u$ per la regola $A \to u$
$a, a \to \varepsilon$ per il terminale a

con Γ che rappresenta l'insieme degli stati finali, raggruppandoli tutti, una stringa e il simbolo iniziale.
Il loop è la fase in cui l'automa sceglie lo stato applicabile.
Prendiamo il seguente esempio:

Trasformiamo la seguente CFG in PDA:

$$S \to aTb \mid b$$
$$T \to Ta \mid \varepsilon$$

Dobbiamo metterci 3 stati:
q_{start}, q_{loop}, q_{end}
e successivamente diciamo "se c'è simbolo di input nella pila, poi lo rimuovo":



L'idea quindi è di seguire la leftmost derivation, quindi avremmo:

- ε, S → aTb (prima cosa fatta)
- l'altra transizione possibile di S
- le due transizione possibili di T
- a e b (in pop) perché simboli terminali

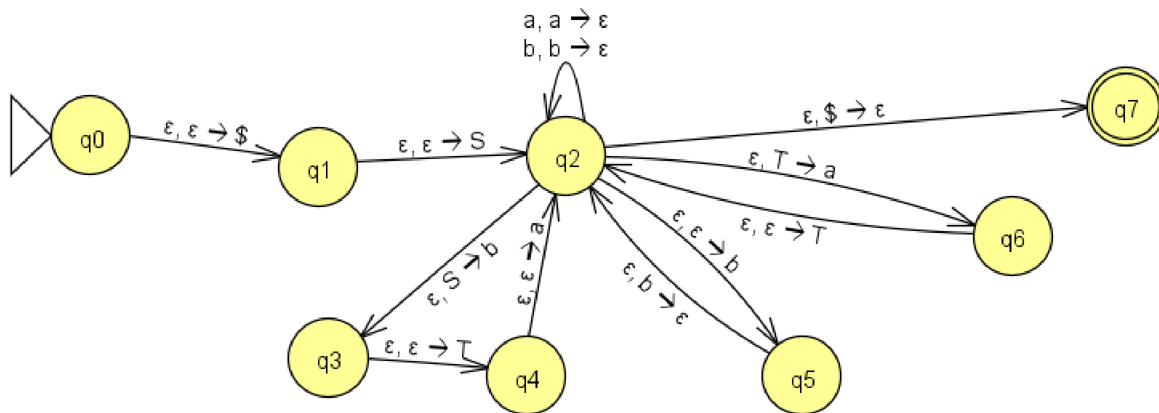
Per esempio scegliamo di attuare una derivazione
S → aTb → aTab → aab

~~S~~
$

~~a~~
T
b
$

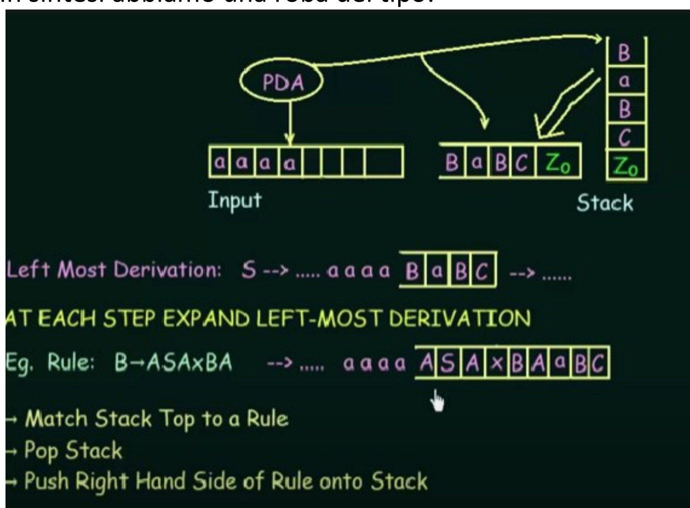
~~T~~
~~a~~
~~b~~
~~$~~

La rappresentazione estesa di come funzionerebbe esattamente questa pila è:

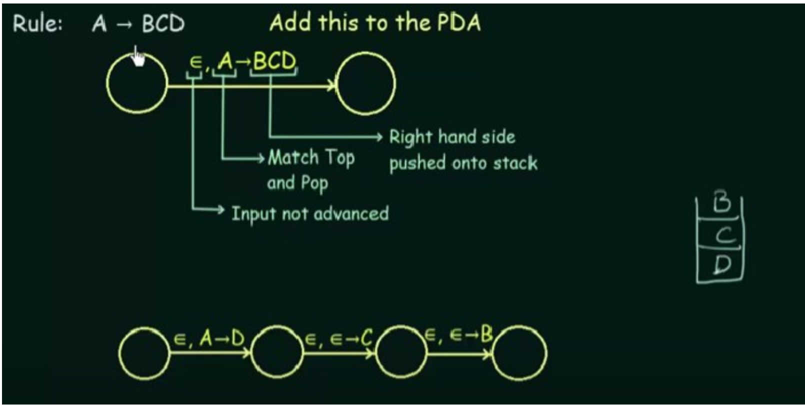


Qui andremmo a inserire il dollaro, S (quindi stati iniziali), avremmo in loop i terminali e verso altri stati delle transizioni che inseriscono le regole nell'ordine contrario (stack, quindi metto bTa per la regola S → aTb), per tutte le non terminali

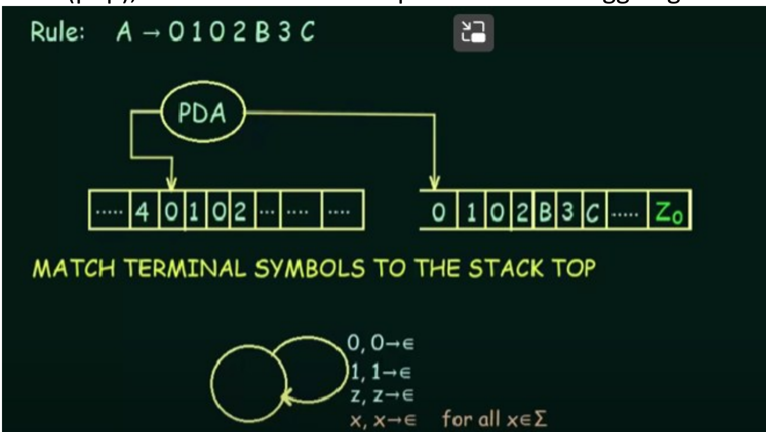
In questo caso, quindi, seguiamo la leftmost derivation, buttiamo dentro (push) tutti i simboli nell'ordine detto dopodiché, sapendo che rimarranno nella pila solo "a" e "b", andremo ad eseguirlo il pop se presenti. In sintesi abbiamo una roba del tipo:



Quindi vogliamo partire sempre da sinistra con le derivazioni, facendo in modo di avere almeno una regola di quel tipo sulla cima dello stack.
Se abbiamo regole composte da più elementi, le spezzettiamo e facciamo *push* in maniera ordinata:



Successivamente, con questa logica, l'elemento che ho buttato dentro per ultimo sarà il primo ad essere tolto (pop), facendo avanzare l'input finché non raggiungiamo la fine dell'input:



Vediamo un altro esempio concreto (sempre trovato su YouTube, convertendo la CFG seguente in PDA).

S → aBc | ab

B → SB | ε

In pratica bisogna introdurre nell'ordine inverso le regole.

Quindi, notiamo che le uniche regole di lettura sono "a,b,c", messe nell'ordine inverso, quindi di rimozione che sarà fattibile dallo stack.

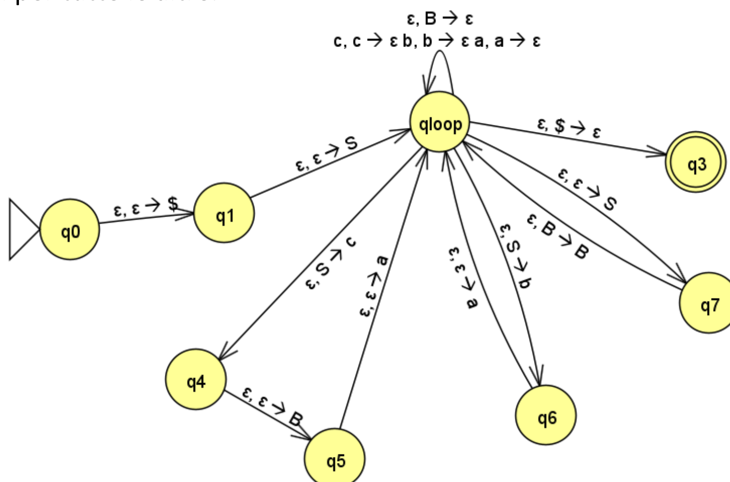
In tutti gli altri casi ragioniamo in questo modo. Prendiamo ad esempio S → aBc.

Prendiamo prima c, quindi ε, S → c, poi mettiamo B, quindi ε, ε → B, dunque mettiamo a, quindi ε, ε → a.

Per tutte le altre regole si procede nella maniera similare.

Similmente per S → ab, avremo prima ε, S → b e successivamente il push di A senza pop, quindi ε, ε → a.

Così per tutte le altre.



A questo punto vogliamo operare la *conversione da PDA a CFG*:

Lemma

Se un è riconosciuto da un PDA, allora è un linguaggio context-free

Idea.

- Abbiamo un PDA P che riconosce il linguaggio
- Mostriamo come trasformare P in una CFG equivalente G

Similmente se una stringa w è accettata dalla pila P, andiamo dallo stato iniziale a quello finale.
Progetteremo quindi una grammatica che fa un po' di più:

- una variabile A_{pq} per ogni coppia di stati p, q di P
- A_{pq} genera tutte le stringhe che portano da p con pila vuota a q con pila vuota

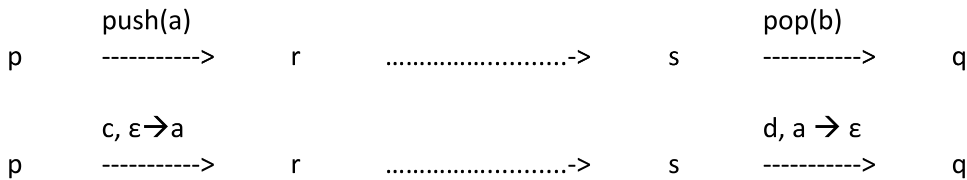
Dobbiamo semplificare la pila P tale che rispetti 3 condizioni:

1. Ha un unico stato accettante q_f
2. Svuota la pila prima di accettare
3. Ogni transizione inserisce un unico simbolo sulla pila (push) oppure elimina un simbolo dalla pila (pop), ma non fa entrambe le cose contemporaneamente

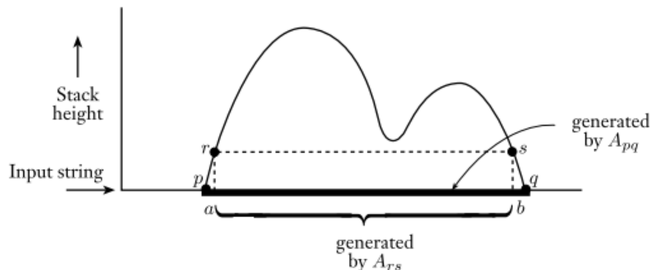
Se valgono queste condizioni, andando da *p* con pila vuota a *q* con pila vuota, dobbiamo avere:

- la prima mossa deve essere un push
- l'ultima mossa deve essere un pop

  $A_{pq} \rightarrow cA_{rs}d$



1. il simbolo inserito all'inizio viene eliminato alla fine:



Concretamente quindi si va da *r* ad *s* mantenendo *a* nella parola.

Case-1:

a, push z ... b, pop z

What strings can be generated by following this path ?

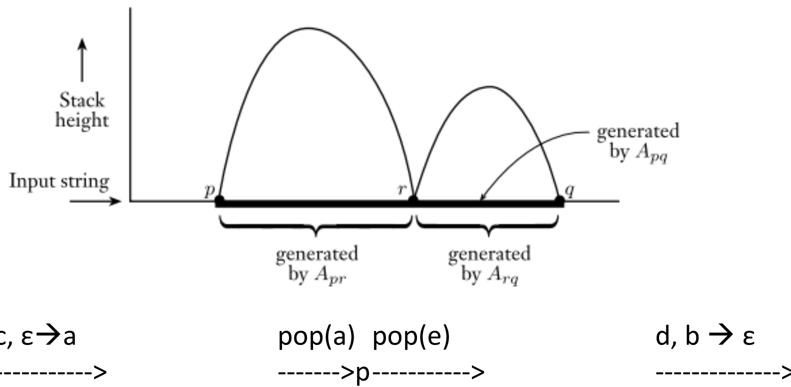
--> "a........b"

$A_{pq} \to a A_{rs} b$

--> This rule will generate exactly those strings.

- il simbolo inserito all'inizio può <u>non</u> essere eliminato (significa che facciamo *pop* di un simbolo non terminale):

2 oppure no:



c, ε→a   pop(a) pop(e)   d, b → ε

p   ----------->   ------->p----------->   -------------->

Case-2:

a, push w ... b, pop z

w gets popped

z gets pushed

What strings can be generated by following this path ?

$A_{pq} \to A_{pr} A_{rq}$

--> This rule will generate exactly those strings.

Formalmente:

- Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_f\})$
- Costruiamo $G = (V, \Sigma, R, A_{q_0, q_f})$ tale che:
  - $V = \{A_{pq} \mid p, q \in Q\}$
  - per ogni $p, q, r, s \in Q$, $u \in \Gamma$ e $a, b \in \Sigma_\varepsilon$, se $\delta(p, a, \varepsilon)$ contiene (r, u) e $\delta(s, b, u)$ contiene (q, ε), aggiungi la regola $A_{pq} \to aA_{rs}b$
  - per ogni $p, q, r \in Q$, aggiungi la regola $A_{pq} \to A_{pr}A_{rq}$
  - per ogni $p \in Q$, aggiungi la regola $A_{pp} \to \varepsilon$

Ecco quindi che avremo una roba del tipo:
$A_{pq} \rightarrow cA_{rs}d \mid A_{pr}A_{rq}$

	a, ε→u			b, u → ε	
p	----------->	r	s	----------->	q

$A_{pq} \rightarrow aA_{rs}b$
$A_{pq} \rightarrow A_{pr}A_{rq}$
$A_{pp} \rightarrow \varepsilon$

Diamo quindi due dimostrazioni induttive, partendo da:

Lemma

Se A_{pq} genera la stringa x, allora x può portare P da p con pila vuota a q con pila vuota

A_{pq} ---->* x   in k passi

Caso base: k=1 $A_{pp} \rightarrow \varepsilon$ regola applicata → p=q   <u>OK</u>

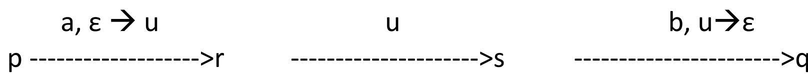
Caso induttivo: suppongo per induzione che il Lemma valga per tutti i valori <= k,
$A_{pq} \rightarrow^* x$ in k+q passi

1) $A_{pq} \rightarrow aA_{rs}b \rightarrow^* x = a\ y\ b$ per ip. induttiva y   r - pila vuota → s - pila vuota

a, ε → u	u	b, u→ε
p ------------------->r	--------------------->s	----------------------->q

1) $A_{pq} \rightarrow A_{pr}A_{rq} \rightarrow^* x = yz$ $A_{pr} \rightarrow^* y$ con <= k passi
$A_{rq} \rightarrow^* z$ con <= k passi

Abbiamo la seconda che ragiona in modo contrario:



Lemma

Se la stringa x può portare P da p con pila vuota a q con pila vuota, allora A_{pq} genera x

P va da p - pila vuota a q - pila vuota con *x* in *k* transizioni
Caso base: k=0   p=q   x=ε $A_{pp} \rightarrow \varepsilon$

Caso induttivo: suppongo ipotesi vera per tutte le computazioni con <= *k* transizioni

1)

a,ε→u		b,u→ε	x=ayb	$A_{rs} \rightarrow^* y$ per hp.ind
p------------------------->r	……………………	s--------------------------->q		

$A_{pq} \rightarrow aA_{rs}b \rightarrow^* ayb \rightarrow x$

2)
p ………………… r ………………… q   x=yz $A_{pr} \rightarrow^* y$, $A_{rq} \rightarrow^* z$, $A_{pq} \rightarrow A_{pr}A_{rq} \rightarrow^* y$, $A_{rq} \rightarrow^* yz=x$

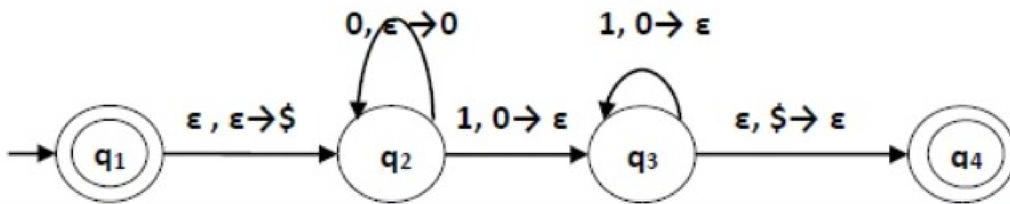
Dunque abbiamo correttamente dimostrato il teorema:

Theorem

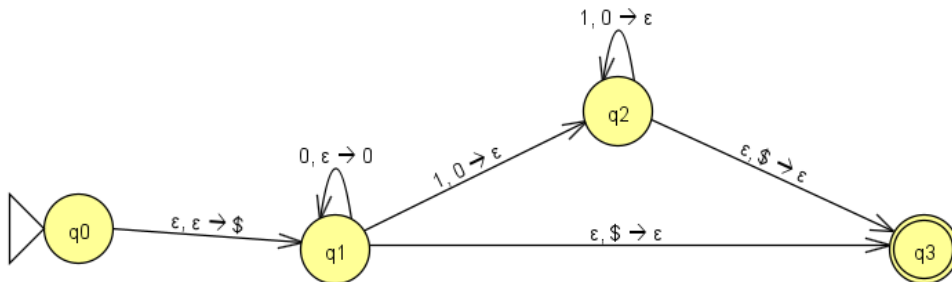
Un linguaggio è context-free se solo se esiste un PDA che lo riconosce

- Sappiamo che un linguaggio è context-free se esiste una CFG che lo genera
- Abbiamo mostrato come trasformare una CFG in un PDA
- E viceversa, come trasformare un PDA in una grammatica

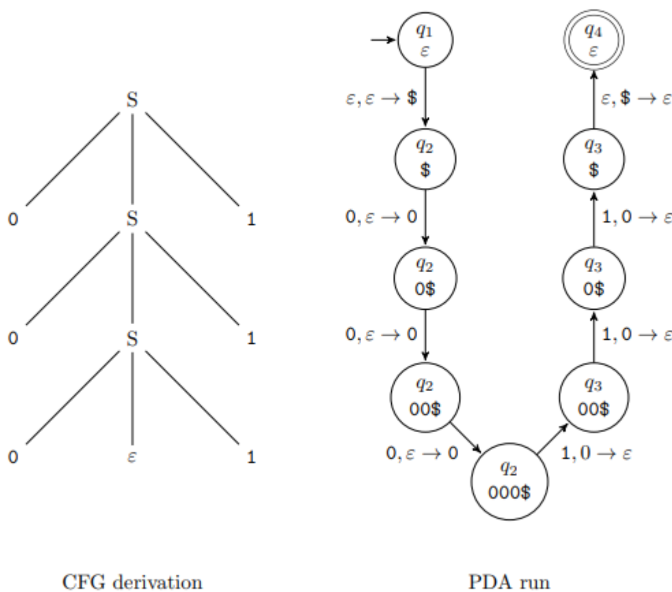
Trasformiamo il PDA per il linguaggio $\{0^n1^n \mid n>=0\}$ in grammatica:



In poche parole mette la transizione che da q2 va a q4 inserendo ε, $ → ε perché potrebbe non esserci nessun simbolo:



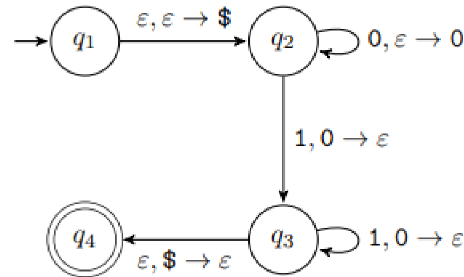
Quindi potremmo avere concretamente una situazione di questo tipo:



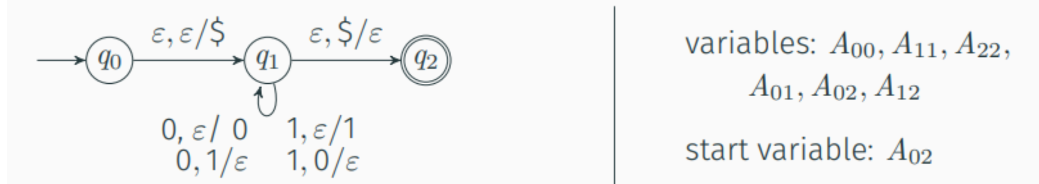
La grammatica prodotta è (riporto l'automa identico al nostro, ma comunque per chiarire col discorso lettere a fianco della CFG):

- si parte considerando i 4 stati che vanno ad ε
- si inseriscono poi tutti gli stati che vanno, per combinazione, tutti gli uni con gli altri (letteralmente è una proprietà commutativa, l'immagine sotto chiarisce)
- nel caso di A_{23} abbiamo l'unione degli stati uscenti
- come ultimo abbiamo la regola dello stato finale

$$
\begin{aligned}
&A_{11} \to \varepsilon \\
&A_{22} \to \varepsilon \\
&A_{33} \to \varepsilon \\
&A_{44} \to \varepsilon \\
&A_{11} \to A_{11}A_{11} \mid A_{12}A_{21} \mid A_{13}A_{31} \mid A_{14}A_{41} \\
&A_{12} \to A_{11}A_{12} \mid A_{12}A_{22} \mid A_{13}A_{32} \mid A_{14}A_{42} \\
&A_{13} \to A_{11}A_{13} \mid A_{12}A_{23} \mid A_{13}A_{33} \mid A_{14}A_{43} \\
&\quad \ldots \\
&A_{42} \to A_{41}A_{12} \mid A_{42}A_{22} \mid A_{43}A_{32} \mid A_{44}A_{42} \\
&A_{43} \to A_{41}A_{13} \mid A_{42}A_{23} \mid A_{43}A_{33} \mid A_{44}A_{43} \\
&A_{44} \to A_{41}A_{14} \mid A_{42}A_{24} \mid A_{43}A_{34} \mid A_{44}A_{44} \\
&A_{23} \to 0A_{22}1 \mid 0A_{23}1 \\
&A_{14} \to \varepsilon A_{23} \varepsilon \\
&S \to A_{14}
\end{aligned}
$$



Altro esempio utile (dei pochi che ho trovato in giro su questa cosa):



productions:

$$
\begin{aligned}
&A_{02} \to A_{01}A_{12} \\
&A_{01} \to A_{01}A_{11} \\
&A_{12} \to A_{11}A_{12} \\
&A_{11} \to A_{11}A_{11} \\
&A_{11} \to 0A_{11}1 \\
&A_{11} \to 1A_{11}0 \\
&A_{02} \to A_{11} \\
&A_{00} \to \varepsilon,\ A_{11} \to \varepsilon, \\
&A_{22} \to \varepsilon
\end{aligned}
$$

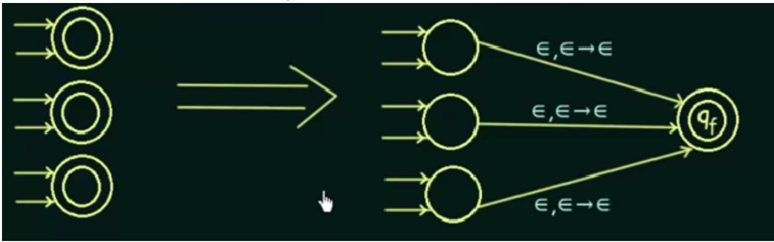
In sintesi quindi:

Given a PDA --> Build a CFG from it

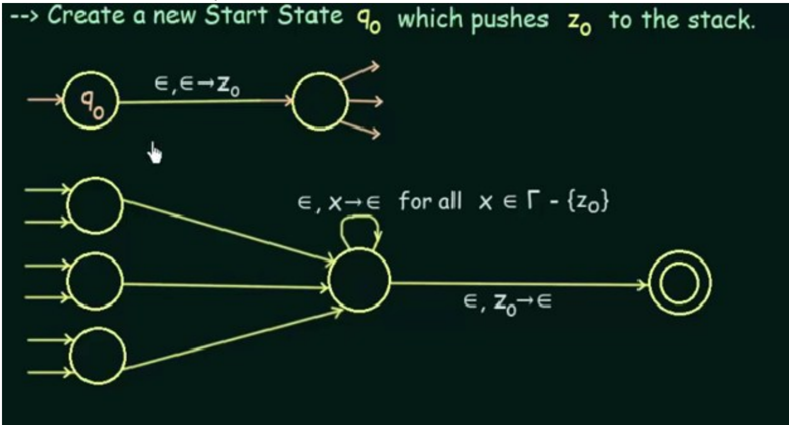
Step 1: Simplify the PDA
Step2: Build the CFG

There will be a Non-Terminal for every pair of states : A_{pq} , A_{qr} , A_{rq_0}, ...
The starting Non-Terminal will be : $A_{q_0q_f}$

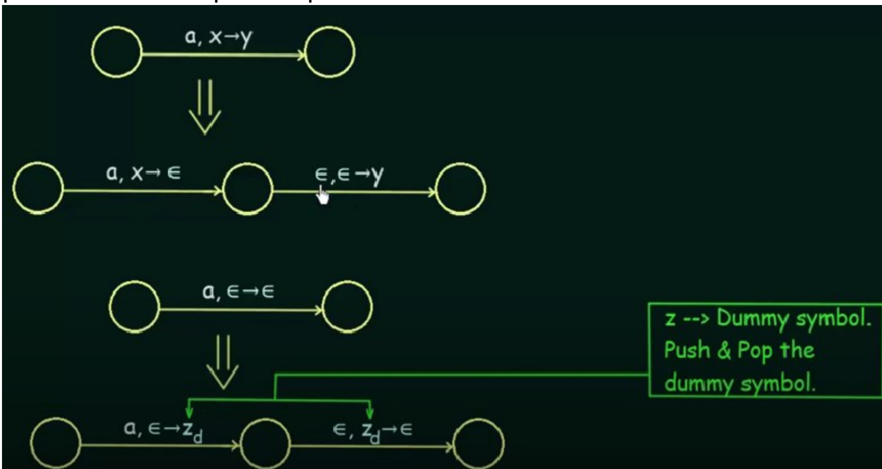
Cominciamo con la semplificazione del PDA, facendo in modo di avere un solo stato finale:



Successivamente prima di accettare il PDA deve svuotare il suo stack:



Attenzione che le transizioni o fanno push oppure pop, ma non devono fare entrambe le cose. Come si vede, non possiamo avere un caso in cui non ci sia pop/push, quindi piazziamo un nuovo simbolo che permetta di fare questa operazione:



Domande Wooclap

Considera la grammatica G per le espressioni aritmetiche:

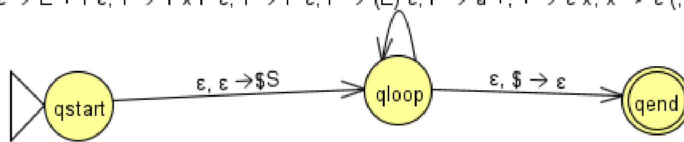
$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid a$$

Costruisci il PDA equivalente alla grammatica.
Quante sono le transizioni *estese* che vanno da q_{loop} a q_{loop}?

ε, ε → T
ε, ε → E + T

ε, T → T x F
ε, T → F
ε, F → (E)
ε, F → a
considerando i 5 simboli terminali
+, + → ε
x, x –> ε
(, (→ ε
),) –> ε
a, a → ε
avremo 6+5=11 simboli.

Quindi se rappresentato sarebbe:



Come va modificato il PDA per poterlo trasformare in grammatica?

Risposta: Deve svuotare la pila prima di accettare

La variabile A_{pq} genera tutte le stringhe x tali che

Risposta: x porta il PDA da “p” con pila vuota a “q” con pila vuota

La regola $A_{pq} \rightarrow aA_{rs}b$ corrisponde al caso

Risposta: Il simbolo inserito nella pila all’inizio viene rimosso alla fine di *x*.

La regola $A_{pq} \rightarrow A_{pr}A_{rq}$ corrisponde al caso

Risposta: Il simbolo inserito nella pila all’inizio viene rimosso prima della fine di *x*.

Linguaggi non context-free

Il pumping lemma dimostra che esistono linguaggi non regolari; vedremo anche per i linguaggi CF esiste una pumping length per cui tutte le stringhe più lunghe possono essere iterate, dimostrando che alcuni linguaggi non sono context-free.
L’enunciato è molto simile ma diverso:

Theorem (Pumping Lemma per Linguaggi Context-free)

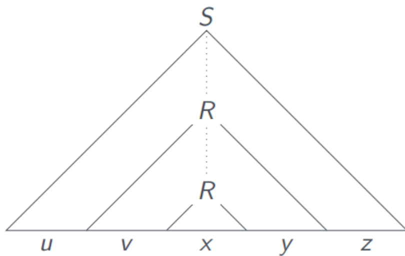
Sia L un linguaggio context-free. Allora

- *esiste una lunghezza $k \geq 0$ tale che*
- *ogni parola $w \in L$ di lunghezza $|w| \geq k$*
- *può essere spezzata in $w = uvxyz$ tale che:*
  1. $|vy| > 0$ *(il secondo o il quarto pezzo non sono la stringa vuota)*
  2. $|vxy| \leq k$ *(il blocco centrale è lungo al max k)*
  3. $\forall i \geq 0$, $uv^ixy^iz \in L$ *(possiamo “pompare” contemporaneamente v e y rimanendo in L)*

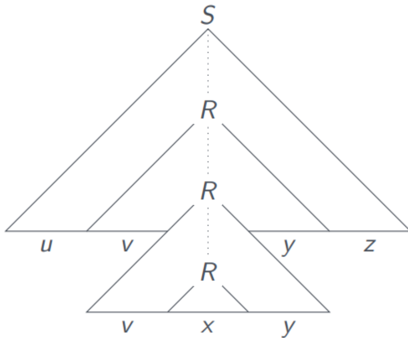
L'idea è di considerare una grammatica G che genera il linguaggio L. Cominciamo prendendo una stringa w "molto lunga" in L, con un albero sintattico "molto alto", quindi esiste un cammino nell'albero che ripete una variabile, replicando il sottoalbero replicando una stringa:

- Consideriamo la grammatica G che genera il linguaggio L
- Prendiamo una stringa w "molto lunga" in L
- L'albero sintattico di w deve essere "molto alto"
- Esiste un cammino nell'albero che ripete una variabile R
- Replicando il sottoalbero di R possiamo "pompare" la stringa rimanendo nel linguaggio

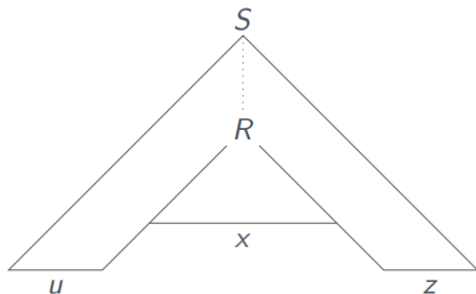
Si ottengono ricorsivamente degli alberi, tali da considerare le ripetizioni dei caratteri, come si vede dividendo la parola in 5 pezzi ripetibili:



avendo il loop che in questo caso non ripete solo il simbolo ma direttamente tutto l'albero sintattico:



(caso uv^ixy^iz)
conseguentemente:



Utilizziamo dunque il PL per dimostrare che il linguaggio *non* sia regolare.

Dunque:

- Ogni CFL soddisfa il Pumping Lemma context-free.
- Un linguaggio che falsifica il Pumping Lemma non può essere context-free:
  - per ogni lunghezza $k \geq 0$
  - esiste una parola $w \in L$ di lunghezza $|w| \geq k$ tale che
  - per ogni suddivisione $w = uvxyz$ tale che:
    1. $|vy| > 0$ (v e y non entrambi vuoti)
    2. $|vxy| \leq k$ (il pezzo centrale è lungo al max k)
  - esiste un $i \geq 0$ tale che $uv^ixy^iz \in L$ (possiamo "pompare" ed uscire da L)

Attenzione!

Esistono linguaggi non context-free che rispettano il Pumping Lemma: $\{a^ib^jc^k \mid i, j, k \text{ tutti diversi tra loro}\}$

Anche qui lo vediamo come ***gioco del PL*** con le seguenti regole:

- L'avversario sceglie la lunghezza k
- Noi scegliamo una parola w
- L'avversario spezza w in $uvxyz$
- Noi scegliamo i tale che $uv^ixy^iz \notin L$
- allora abbiamo vinto

Prendiamo per esempio JFLAP, cliccando "*Computer Goes First*":

$L = \{a^nb^nc^n : n \geq 0\}$ Select

Similmente cerchiamo di vincere prendendo una lunghezza maggiore per ogni suddivisione possibile scelta dal PC:

$L = \{a^nb^nc^n : n \geq 0\}$ Context-Free Pumping Lemma

Objective: Prevent the computer from finding a valid partition.

Clear All | Explain

1. I have selected a value for m, displayed below.

8

2. Please enter a possible value for w and press "Enter".

Voglio mettermi in una situazione simile a quella che conosciamo:

w = uvxyz
|vy| > 0
|vxy| <= k

Scelta la parola:

2. Please enter a possible value for w and press "Enter".
aaaaaaaabbbbbbbbcccccccc

3. I have decomposed w into the following...
U = aaaaaaaabbbbbbbbcccccc; V = cc; X = ε; Y = ε; Z = ε

4. Please enter a possible value for i and press "Enter".
i: pumped string:

mettiamo un valore che faccia uscire dal linguaggio:

4. Please enter a possible value for i and press "Enter".
: 2 pumped string: aaaaaaaabbbbbbbbcccccccccc

5. Animation

w = aaaaaaaabbbbbbbbcccccc cc _ _ _ (u, v, x, y, z)

$uv^2xy^2z = a^8b^8c^{10}$ = aaaaaaaabbbbbbbbcccccccccc is NOT in the language. YOU WIN!

Step Restart

Supponiamo che L sia CFL.
Esiste una lunghezza del pumping "k"
Considero la parola $w = a^kb^kc^k \in L$ $|w| > k$
Data una qualsiasi suddivisione w=uvxyz tale che |vy| > 0 e |vxy| <= k.

Non dicendo nulla su "u" significa che possiamo sceglierla ovunque nella parola, essendo a cavallo tra le "a" e le "b", oppure all'inizio, se non alla fine. Non possiamo avere contemporaneamente "a", "b", "c".

aaa……………..abb………………..bcc…………….c
k

$uv^2xy^2z \notin L$ perché solo due lettere su tre aumentano di numero.
Andiamo quindi a JFLAP stavolta con "*You go first*".

$L = \{a^nb^nc^n : n \geq 0\}$ **Context-Free Pumping Lemma**

Objective: Find a valid partition that can be pumped.
Clear All Explain

1. Please select a value for m in Box 1 and press "Enter".

Cases:

#	Description

Proviamo a prendere:
n=1
w=ab

u=ε, z= ε, x=ε
v=a |vy| > 0 ✓
y=b |vxy| <= 2

$uv^ixy^iz = a^ib^i$

Noi scegliamo 4:

2. I have selected w such that |w| >= m. It is displayed in Box 2.
aaaabbbb

3. Select decomposition of w into uvxyz.
u: aaa |u|: 3
v: a |v|: 1
x: |x|: 0 Set uvxyz
y: b |y|: 1
z: bbb |z|: 3
a a a a b b b b

4. I have selected i to give a contradition. It is displayed in Box 4.
i: 2 pumped string: aaaaabbbbb

5. Animation
u v x y z
w = aaa a _ b bbb

$uv^2xy^2z = a^5b^5$ = aaaaabbbbb is in the language. YOU WIN!

Step Restart

Add Replace Show Delete
List Clear Done?

L'idea di fondo quindi è pompare la stringa tale da essere sempre rappresentabile come albero, ricorsivamente radicato in se stesso, diventando potenzialmente molto alto.

Idea di dimostrazione del Pumping Lemma per linguaggi context-free:

- Consideriamo la grammatica G che genera il linguaggio L
- Prendiamo una stringa w "molto lunga" in L
- L'albero sintattico di w deve essere "molto alto"
- Esiste un cammino nell'albero che ripete una variabile R
- Replicando il sottoalbero di R possiamo "pompare" la stringa rimanendo nel linguaggio

Per passare dall'idea alla dimostrazione dobbiamo stabilire cosa vuol dire che la stringa è molto lunga e perché l'albero sintattico è molto alto

Ragioniamo quindi sulla proprietà generica degli alberi, tali da rispettare un numero di nodi logaritmico per livello (classica proprietà degli alberi binari):

- Sia G una grammatica per il linguaggio L
- Sia b il numero massimo di simboli nel lato destro delle regole
- In un albero sintattico, ogni nodo avrà al massimo b figli:
  - al più b foglie per un albero di altezza 1
  - al più b^2 foglie per un albero di altezza 2
  - al più b^h foglie per un albero di altezza h
- Un albero di altezza h genera una stringa di lunghezza minore o uguale a b^h

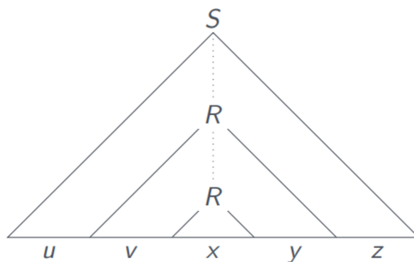
- Viceversa: una stringa di lunghezza maggiore o uguale a $b^h + 1$ richiede un albero sintattico di altezza maggiore di h

Similmente andremo a prendere una pumping length simile al numero di nodi, prendendo il più piccolo albero sintattico ed il cammino più lungo, ripetendo una variabile andando verso il basso (ricorsivamente), per equiparare il numero di variabili della grammatica:

- Prendiamo come lunghezza del pumping $k = b^{|V|+1}$ ($|V|$ numero di variabili in G)
- Presa una stringa $w \in L$, se $|w| \geq k$ allora ogni albero sintattico per w deve avere altezza maggiore o uguale a $|V| + 1$
- Prendiamo più piccolo albero sintattico τ per w
- Prendiamo il cammino più lungo in τ: sarà di lunghezza $\geq |V| + 2$
- Quindi ci sono almeno $|V| + 1$ variabili nel cammino
- Qualche variabile R si ripete nel cammino
- Scegliamo R in modo che si ripeta nei $|V| + 1$ nodi più in basso nel cammino

Operiamo quindi la suddivisione di R in w=uvxyz:

- il sottoalbero più in alto genera vxy
- il sottoalbero più in basso genera x



Operiamo quindi il pumping su "w":

- possiamo sostituire i due sottoalberi tra loro, ottenendo di nuovo un albero sintattico corretto:
  - sostituire ripetutamente il più piccolo con il più grande ci dà uv^ixy^iz per ogni $i > 1$
  - sostituire il più grande con il più piccolo ci dà uxz
- in tutti i casi la nuova stringa appartiene a L
- rimane da dimostrare:
  - che $|vy| > 0$ (v e y non possono essere entrambi la parola vuota)
  - che $|vxy| \leq k$

Se esiste, con classica dimostrazione per assurdo, un albero più piccolo nella costruzione ricorsiva verso il basso, non può esisterne uno più piccolo.

- supponiamo che $v = \varepsilon$ e $y = \varepsilon$
- allora se sostituiamo il sottoalbero più grande con il più piccolo otteniamo di nuovo w:

$$w = uvxyz = u\varepsilon x\varepsilon z = uxz$$

- Assurdo: avevamo scelto τ come l'albero sintattico più piccolo!

Quindi:

- l'occorrenza più in alto di R genera vxy
- avevamo scelto le occorrenze tra i $|V|+1$ nodi più in basso
- il sottoalbero che genera vxy è alto al massimo $|V|+1$
- quindi può generare una stringa di lunghezza al più $b^{|V|+1}=k$

FINE!

<u>*Domande Wooclap*</u>

Quali dei seguenti linguaggi sono context-free? Puoi scegliere più di una risposta

- $\{a^n b^n \mid n \geq 0\}$
- $\{a^n b^n c^n \mid n \geq 0\}$
- $\{a^n b^j a^n b^j \mid n, j \geq 0\}$
- $\{a^n b^j a^j b^n \mid n, j \geq 0\}$
- $\{ww \mid w \in \{0,1\}^*\}$
- $\{ww^R \mid w \in \{0,1\}^*\}$

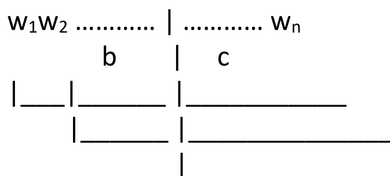
1) Si
2) No
3) No
4) No
5) No
6) Si

Si discutono soluzioni di esercizi:

suffix(L) = {v | uv ∈ L}

Sia G la grammatica che genera L.
Assumiamo che G sia in forma normale di Chomsky.
Avremo quindi regole del tipo:
A → BC
D → d

Siccome c'è una concatenazione, un pezzo della parola sarà generato da "b" e un altro da "C"



A' suffissi delle parole generate da A
A' → A|B'C|C|C'|ε

Quindi quello che viene adesso è la variabile iniziale
D' → d|ε

Scritto da Gabriel

S' è la variabile iniziale sse S è variabile iniziale di G

L'esercizio dei *prefix* inverte la stessa cosa.

Sia A un linguaggio, e sia $DROPOUT(A)$ come il linguaggio contenente tutte le stringhe che possono essere ottenute togliendo un simbolo da una stringa di A:

$$DROPOUT(A) = \{xz \mid xyz \in A \text{ dove } x, y \in \Sigma^* \text{ e } y \in \Sigma\}.$$

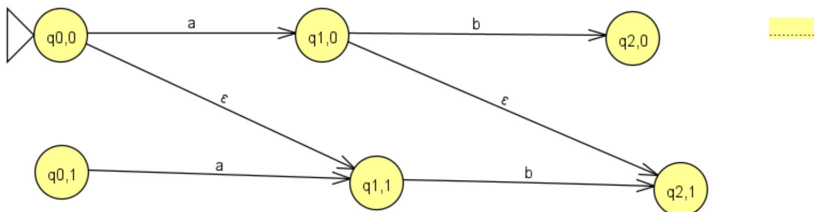
Mostrare che la classe dei linguaggi regolari è chiusa rispetto all'operazione $DROPOUT$, cioè che se A è un linguaggio regolare allora $DROPOUT(A)$ è un linguaggio regolare.

Quindi:
{xz | xyz ∈ A, x, z ∈ Z*, y ∈ Σ}
Sia M=(Q, Σ ,δ, q_o, F) un DFA che riconosce A.
M'=(Q', Σ , δ', q_o, F')

In pratica si saltano a random tutti gli elementi, andando in uno, un altro o un altro ancora, senza un ordine definito. Per ognuno deve ricordarsi che ha saltato quel carattere (non ne salta più di uno):
$w_1 w_2$ ....w_r
$q_0 q_1$..............q_n

Q'=Q x {0,1}   (q, 0)   "non ho saltato"
q' = (q_0, 0)   (q,1)   "ho saltato"

δ' ((q,0)a)={(δ(q_0,a),0)}
δ((q,0)ε)={(p,1)| δ(q,a)=p   per qualche a ∈ Σ
δ' ((q,1)a)={(δ(q,a),1)}

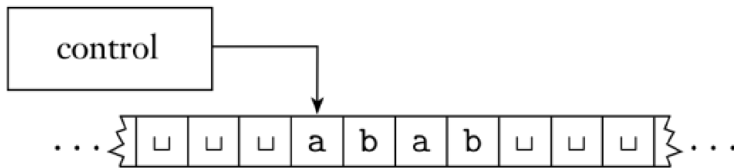


....

F'={(q1,1)|q_0∈ F}

Macchine di Turing

La *macchina di Turing* si pone come modello creato da Turing per creare una macchina illimitata e senza restrizioni; non tutti i problemi sono da esse risolvibili, in quanto alcuni di questi vanno oltre a capacità di una macchina. Essa di fatto è un automa a stati finiti, ma dotata di una memoria illimitata fatta a nastro, con una testina che legge/scrive simboli sul nastro, *scrivendo* sul nastro per memorizzare informazioni, *leggendo* i dati muovendo la testina ovunque sul nastro.
Essa dispone di stati speciali per *accettare* e *rifiutare*.



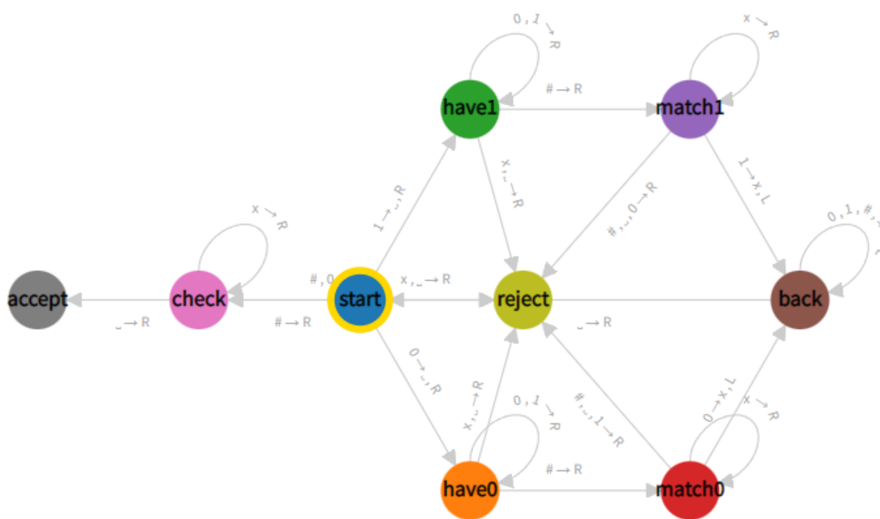
Vediamo un primo esempio di macchina di Turing per rappresentare come linguaggio:

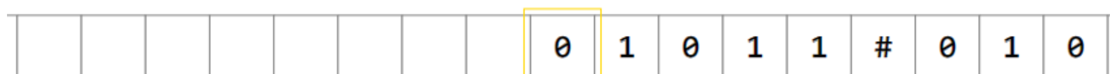
- Costruiamo una macchina di Turing per il linguaggio

$$B = \{w\#w \mid w \in \{0,1\}^*\}$$

- M_2 deve accettare se l'input sta in B, e rifiutare altrimenti

Immaginiamo che la stringa sia molto lunga (es. un milione di simboli); ovviamente non sarà mai memorizzabile completamente. Controlliamo quindi se le stringhe sono uguali, andando un carattere uno alla volta, fino al cancelletto. Se il carattere successivo carattere al cancelletto va bene, si salta avanti/indietro al cancelletto. A questo punto, se combaciano tutti i caratteri, si procede continuativamente a zigzag. Di fatto controllo che il carattere sia marcato o meno; se l'ho controllato vado avanti.
L'esempio di macchina è:





La descrizione ad alto livello è la seguente:

- M_2 deve accettare se l'input sta in B, e rifiutare altrimenti
- M_2 = "Su input w:
  1. Si muove a zig-zag lungo il nastro, raggiungendo posizioni corrispondenti ai due lati di # per controllare se contengono lo stesso simbolo. In caso negativo, o se non trovi #, rifiuta. Barra gli elementi già controllati.
  2. Se tutti i simboli a sinistra di # sono stati controllati, verifica i simboli a destra di #. Se c'è qualche simbolo ancora da controllare rifiuta, altrimenti accetta."

Se a destra ci sono simboli ancora non barrati, da quella parte la stringa è più lunga della metà precedente. Rifiuto fin quando tutti i simboli non sono stati barrati e controllati.
Vediamo l'esempio completo da https://turingmachine.io/ nell'esempio "equal strings"
Semplicemente prende un simbolo (0 oppure 1) dalla parte prima del # e poi se lo trovo nella parte dopo il # lo segna come barrato, torna all'inizio e ripaerte eseguendo la stessa operazione.

Le macchine di Turing (le chiamerò TM da ora in poi):

- possono leggere/scrivere sul nastro
- possono muoversi sia a dx/sx
- il nastro è infinito
- gli stati di rifiuto/accettazione hanno effetto immediato

Segue la definizione formale:

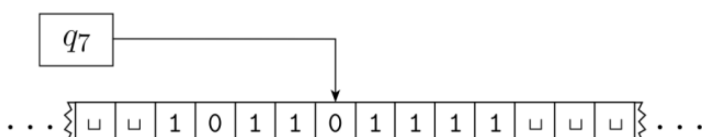
Una Macchina di Turing (o Turing Machine, TM) è una tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:

- Q è l'insieme finito di stati
- Σ è l'alfabeto di input che non contiene il simbolo blank $\sqcup$
- Γ è l'alfabeto del nastro che contiene $\sqcup$ e Σ
- $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$ è la funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $q_{accept} \in Q$ è lo stato di accettazione
- $q_{reject} \in Q$ è lo stato di rifiuto (diverso da q_{accept})

Per descrivere nel dettaglio le macchine di Turing dobbiamo sapere la *configurazione* (tramite stato corrente, posizione della testina e contenuto del nastro). Da questa possiamo sapere *la prossima mossa*. Esse sono rappresentate da una tripla *uqv*:

- q è lo stato corrente
- u è il contenuto del nastro prima della testina
- v è il contenuto del nastro dalla testina in poi
- la testina si trova sul primo simbolo di v

Rappresentiamo dunque in figura $1011q_701111$:



- La configurazione C_1 produce C_2 se la TM può passare da C_1 a C_2 in un passo
- Se $a, b, c \in \Gamma$, $u, v \in \Gamma^*$ e q_i, q_j sono stati, allora:

$$uaq_ibv \text{ produce } uq_jacv \qquad \text{se } \delta(q_i, b) = (q_j, c, L)$$
$$uaq_ibv \text{ produce } uacq_jv \qquad \text{se } \delta(q_i, b) = (q_j, c, R)$$

Questa descritta, anche seguendo il libro, è la configurazione legale della TM.

- La configurazione iniziale con input w è q_0w
- In una configurazione di accettazione lo stato è q_{accept}
- In una configurazione di rifiuto lo stato è q_{reject}
- Le configurazioni di accettazione e rifiuto sono configurazioni di arresto

Volendo dare una definizione induttiva:

- una TM M accetta l'input w se esiste una sequenza di configurazioni $C_1, C_2, \dots, C_k$ tale che:
  - C_1 è la configurazione iniziale con input w
  - ogni C_i produce C_{i+1}
  - C_k è una configurazione di accettazione
- Linguaggio riconosciuto da M: insieme delle stringhe accettate da M

Diamo quindi la seguente definizione di linguaggio *Turing-riconoscibile* (chiamato anche *linguaggio ricorsivamente enumerabile*):

Definition

Un linguaggio è Turing-riconoscibile (o anche ricorsivamente enumerabile) se esiste una macchina di Turing che lo riconosce.

- Se forniamo un input ad una TM, ci sono tre risultati possibili:
  - la macchina accetta
  - la macchina rifiuta
  - la macchina va in loop e non si ferma mai
- la TM può non accettare sia rifiutando che andando in loop

Diciamo quindi che una TM che termina sempre si chiama *decisore* ed un decisore *decide* se un linguaggio se lo riconosce.

Definition

Un linguaggio è Turing-decidibile (o anche ricorsivo) se esiste una macchina di Turing che lo decide.

Se un linguaggio è Turing-decidibile, è anche Turing-riconoscibile; non vale viceversa.

Vediamo quindi una serie di esempi di macchine di Turing (oggi descritte nel dettaglio dal prof, le prossime volte no altrimenti sarebbe lunga).

TM che decide il linguaggio di tutte le stringhe di 0 la cui lunghezza è una potenza di 2:

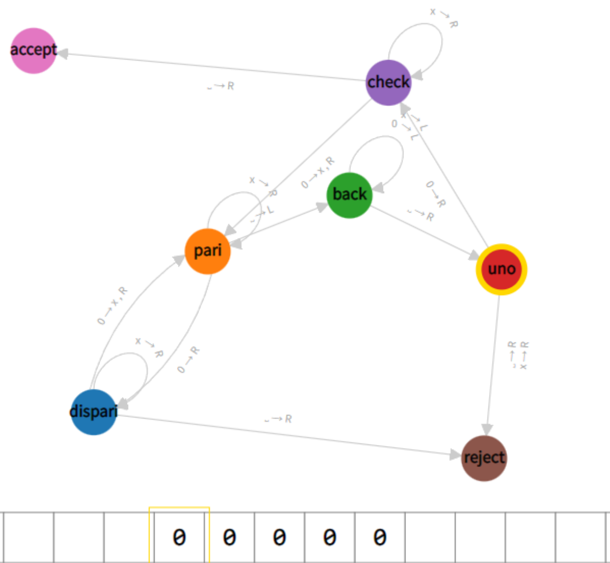
$$A = \{0^{2^n} \mid n \geq 0\}$$

Il funzionamento è il seguente:

M_1 = "su input w:

1. Scorri il nastro da sinistra a destra, cancellando ogni secondo 0
2. Se il nastro conteneva un solo 0, accetta
3. Se il nastro conteneva un numero dispari di 0, rifiuta
4. Ritorna all'inizio del nastro
5. Vai al passo 1."

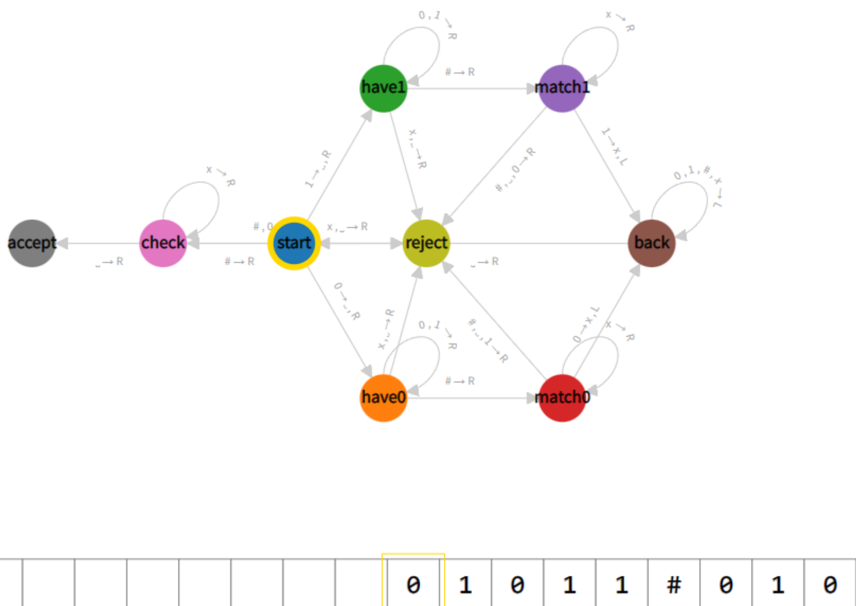
L'esempio della macchina è:



Andiamo poi all'esempio discusso all'inizio

TM che decide il linguaggio:

$$B = \{w\#w \mid w \in \{0,1\}^*\}$$



Segue poi l'esempio di una TM che esegue alcune operazioni aritmetiche:

TM che esegue operazioni aritmetiche. Decide il linguaggio:

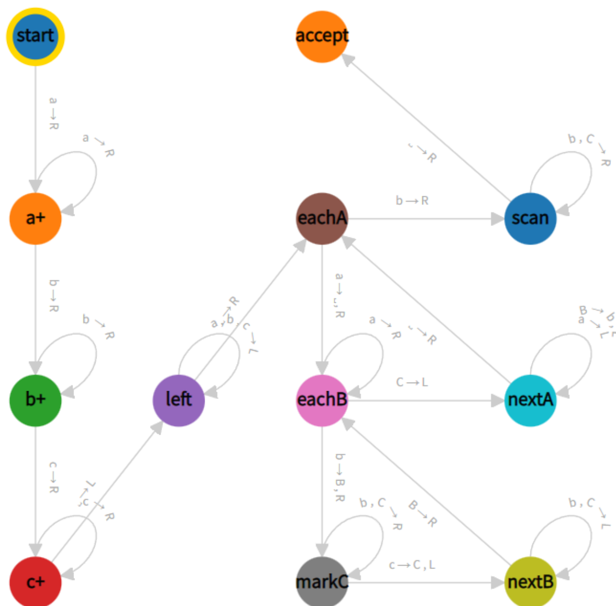
$$C = \{a^i b^j c^k \mid k = i \cdot j \text{ e } i, j, k \geq 1\}$$

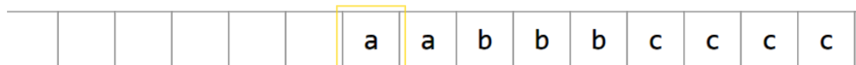
Il ragionamento è di vedere la moltiplicazione con un insieme di somme e per ogni "a" elimino un numero di "c" pari al numero di "b", facendo zig-zag tra "b" e "c", fino alla fine delle "b".
Se rimangono ancora "b" non barrate rifiuto, altrimenti devo ripristinare le "b" barrate e ripeto finché ci sono ancora "a" da barrare.
Più sinteticamente (qui a destra):

$M_3 =$ "su input w:

1. Scorri il nastro da sinistra a destra e controlla se l'input sta in $a^+b^+c^+$. Rifiuta se non lo è.
2. Ritorna all'inizio del nastro
3. barra una a e scorri a destra fino a trovare una b. Fai la spola tra b e c, barrando le b e le c fino alla fine delle b. Se tutte le c sono barrate e rimangono ancora b, rifiuta
4. Ripristina le b barrate e ripeti 3 finché ci sono a da barrare.
5. Quanto tutte le a sono barrate, controlla se tutte le c sono barrate: se si accetta, altrimenti rifiuta."

L'esempio di turingmachine.io è il seguente:





Un esempio di una parola che non accetta (si blocca su *markC*) sarebbe: aabbbccccc

Le TM possono anche scorrere elementi e capire se essi siano uguali/diversi tra di loro.

TM che risolve il problema degli elementi distinti. Prende in input una sequenza di stringhe separate da # e accetta se tutte le stringhe sono diverse. Decide il linguaggio:

$$D = \{\#x_1\#x_2\# \cdots \#x_\ell \mid x_i \in \{0,1\}^* \text{ e } x_i \neq x_j \text{ per ogni } i \neq j\}$$

Possiamo ricordarci dove erano 0/1 usando simboli speciali, per poter capire la diversità (ad esempio marcare gli elementi con una barra), andando avanti/indietro finché i simboli sono uguali. Quando le stringhe sono diverse, torniamo indietro del tutto smarcando tutti i simboli e avanziamo (ad esempio, dopo x1/x2, vado ad x1/x3).

La descrizione completa è:

M_4 = "su input w:

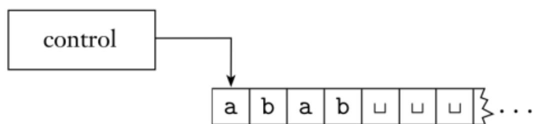
1. Mette un segno sul simbolo del nastro più a sinistra. Se è un blank, accetta. Se è un #, continua con 2. Altrimenti, rifiuta.
2. Scorre a destra fino al successivo # e vi mette sopra un secondo segno. Se nessun # viene trovato, allora era presente solo x_1: accetta.
3. Procede a zig-zag confrontando le due stringhe a destra dei # segnati. Se sono uguali, rifiuta.
4. Sposta il segno più a destra sul successivo # alla sua destra. Se non trova nessun #, sposta il segno più a sinistra sul successivo # alla sua destra, e sposta il segno più a destra sul successivo #. Se non c'è un # dopo il segno più a destra, allora tutte le stringhe sono state confrontate: accetta.
5. Vai alla fase 3."

Concludendo:
- i 4 linguaggi visti cono decidibili
- tutti i linguaggi Turing-decidibili sono anche Turing-riconoscibili
- i linguaggi A,B,C,D sono anche Turing-riconoscibili

Altri esercizi su questo disponibili a: https://elearning.unipd.it/math/mod/url/view.php?id=46301

Varianti delle macchine di Turing

Esistono definizioni alternative delle macchine di Turing e sono definite varianti.
Le varianti *ragionevoli* riconoscono la stessa classe di linguaggio e rappresentano un modello *robusto*, aggiungendo o meno funzionalità alla macchina.
Un esempio di variante semplice è la *macchina a nastro semi-infinito*, nell'esempio solo verso destra:



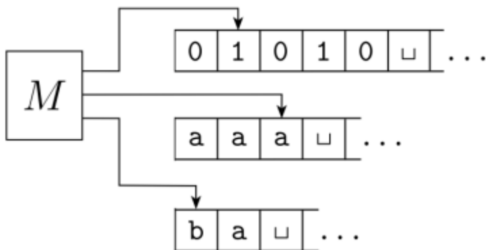
L'input si trova sulla prima posizione del nastro, partendo da quella più a sinistra. Se M tenta di spostare la testina a sinistra se si trova sulla prima cella, allora rimane ferma.
Diamo quindi il seguente teorema:

Theorem

1. *Per ogni TM a nastro semi-infinito esiste una TM a nastro infinito equivalente.*
2. *Per ogni TM a nastro infinito esiste una TM a nastro semi-infinito equivalente.*

Posso quindi, una volta che ho finito il contenuto a sinistra, aggiungere un simbolo che indica "ho finito" e continuo a destra, che non ha una fine. Questa è la condizione di *TM a nastro semi-infinito*. È chiaro quindi come finché non si arrivi ad uno stato accettante, la computazione continui sempre all'infinito, spiegando l'equivalenza.

Similmente, per la seconda casistica, l'operazione svolta è una sorta di shift; immaginando di avere due nastri infiniti, uno a sinistra ed uno a destra, si marca sempre la fine di uno dei due con un simbolo speciale. Assumendo controllo finito, vi saranno due stati finiti rappresentanti. Si scambiano continuativamente le transizioni tra destra e sinistra, fondendo di volta in volta le transizioni e scambiandole in maniera speculare. Per ogni stato esistente si aggiunge un set di transizioni, affinché una volta raggiunto un certo simbolo finale, gli stati vengono riscambiati con la controparte speculare. Questo avviene in maniera continuativa e possibilmente infinita. La differenza finale è appunto la presenza di uno stato accettante in una delle due parti. Quando esso viene raggiunto, si considera semi-infinita, completando la prova.



Abbiamo anche le macchine di Turing con più nastri, definite come *macchine multinastro*, con una TM con "k" nastri semi-finiti, con tutti gli altri nastri vuoti all'inizio della computazione. Con le *k* testine di lettura e scrittura, input sul simbolo 1, ad ogni passo scrive e si muove simultaneamente su tutti i nastri.

Formalmente la funzione di transizione è così formata:

- funzione di transizione:

$$\delta : Q \times \Gamma^k \mapsto Q \times \Gamma^k \times \{L, R\}^k$$

$$\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, L, R, \dots, L) :$$

- se lo stato è q_i e le testine leggono $a_1, \dots, a_k$
- allora scrivi $b_1, \dots, b_k$ sui k nastri
- muovi ogni testina a sinistra o a destra come specificato

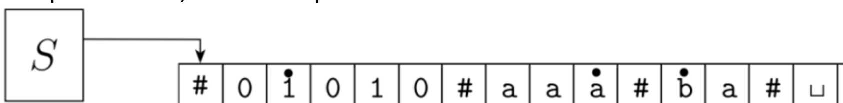
che semplicemente dice questo; ho tra i tre nastri dei simboli separatori e la funzione di transizione indica come muoversi su ciascuno di questi nastri.
Esempio completo:

Se nella macchina multinastro $\delta(q, 1, a, b) = (p, 0, a, a, L, R, R)$, quale sarà il contenuto del primo nastro virtuale dopo la simulazione della transizione?

La funzione di transizione mi dice che io vado da "q" a "p" con stati possibili "a" e "b" con simbolo 1.
Poi ho:

- primo nastro con 0/L
- secondo nastro con a/R
- terzo nastro con a/R

Nel primo caso, con M in queste condizioni:



avrò i 3 nastri in questo stato:

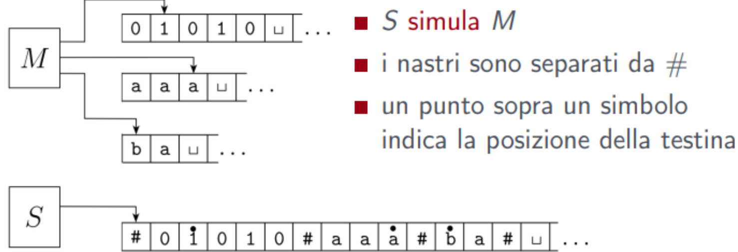
- primo nastro con 0^*1010 (quindi mi sposto con 0 a sx)
- secondo nastro con $aaa_^*$ (quindi mi sposto a dx dove non ci sono simboli)
- terzo nastro con a/R (perché il nastro non finisce a destra e dalla descrizione del linguaggio avrò della "a" a destra e dunque diventa aa^*)

Da qui poi si ha l'*equivalenza tra le macchine multinastro a singolo nastro*:

Theorem

Per ogni TM multinastro esiste una TM a singolo nastro equivalente.

L'idea è di fornire una *simulazione dell'avanzamento degli stati* affinché essi siano separati da simboli e si indichi quando un certo simbolo è stato precedentemente trovato.



In generale S accetta la parola "w" se accetta tutti i suoi input.

1) Si vuole scrivere la configurazione iniziale sul nastro

1 Inizializza il nastro per rappresentare i k nastri:

$$\#\dot{w}_1 w_2 \dots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \dots \#$$

2) Poi inizio a simulare la computazione, dove M cerca di capire qual è la transizione e quali sono i simboli puntati dalla stessa

2 Per simulare una mossa di M, scorri il nastro per determinare i simboli puntati dalle testine virtuali

3) Si fa poi un secondo passaggio del nastro per aggiornare le posizioni

3 Fai un secondo passaggio del nastro per aggiornare i nastri virtuali secondo la funzione di transizione di M

4 Se S sposta una testina virtuale a destra su un $\#$, allora M ha spostato la testina sulla parte vuota del nastro. Scrivi un $\sqcup$ e sposta il contenuto del nastro di una cella a destra

5 Ripeti da **2**

Diciamo quindi anche che:

Un linguaggio è Turing-riconoscibile se e solo se esiste una macchina di Turing multinastro che lo riconosce.

concludendo sulla base della costruzione precedente:

$\Rightarrow$ Un linguaggio è Turing-riconoscibile se è riconosciuto da una TM con un solo nastro, che è un caso particolare di TM multinastro

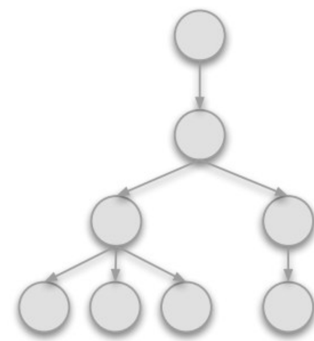
Passiamo quindi alle _macchine di Turing non deterministiche:_

- Una TM non deterministica ha più strade possibili durante la computazione
- Consideriamo macchine con un solo nastro semi-infinito
- La funzione di transizione è:

$$\delta : Q \times \Gamma \mapsto 2^{(Q \times \Gamma \times \{L,R\})}$$

- la computazione è un albero che descrive le scelte possibili
- la macchina accetta se esiste un ramo che porta allo stato di accettazione

Ogni nodo dell'albero rappresenta un'idea di computazione ed ognuno dei nodi dipende da un certo numero di possibilità. Quando almeno uno dei rami trova uno stato di accettazione, allora la macchina accetta una certa parola. Dunque, vogliamo capire come esaminare l'albero, se *in ampiezza* oppure *in profondità*.



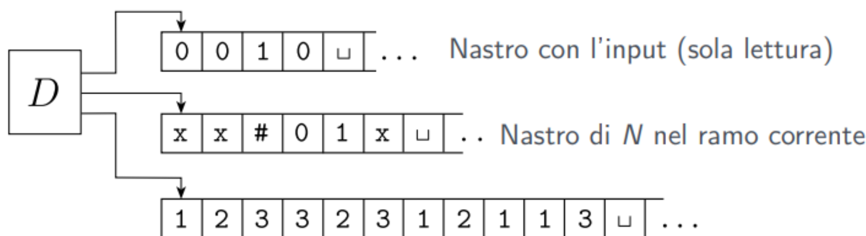
La strategia in profondità ha problemi perché potrebbe rimanere incastrata nel ramo infinito, senza accorgersi che potrebbe andare in un ramo accettante. Usa quindi la visita _in ampiezza (breadth-first)_, trovando in uno dei livelli l'accettazione.

Theorem

Per ogni TM non deterministica N esiste una TM deterministica D equivalente.

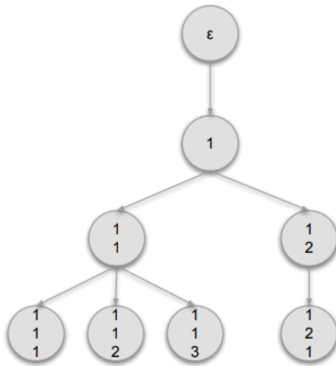
L'idea è di semplificare la computazione creandola su più nastri, con il primo nastro a sola lettura, il secondo nastro che indica che la macchina può costruire N nel ramo corrente, il terzo nastro che tiene traccia delle scelte sulla base dell'indirizzo del nodo corrente.

Idea:



Il terzo nastro tiene traccia delle scelte non deterministiche

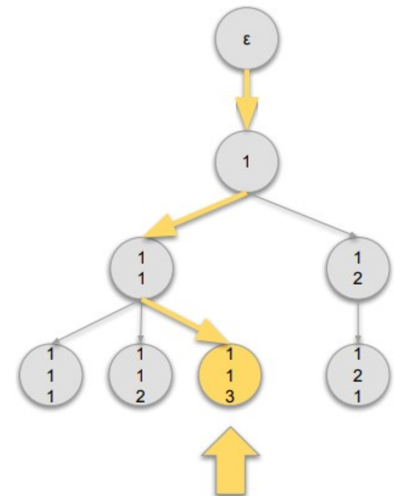
Nel terzo nastro per ogni nodo si assegna un *indirizzo*, corrispondente alla stringa dell'alfabeto della parte che considera il maggior numero di figli dei nodi dell'albero.



- Ad ogni nodo viene assegnato un indirizzo: una stringa sull'alfabeto $\Gamma_b = \{1, 2, \ldots, b\}$, dove b è il massimo numero di figli dei nodi dell'albero

In questo modo possiamo rappresentare tutti i nodi dell'albero, considerando tutti gli indirizzi composti da un certo numero di simboli (prima 1, poi 2, poi 3, così via per profondità).

- Ad ogni nodo viene assegnato un indirizzo: una stringa sull'alfabeto $\Gamma_b = \{1, 2, \ldots, b\}$, dove b è il massimo numero di figli dei nodi dell'albero
- Il nodo 113 si raggiunge prendendo il primo figlio della radice, seguito dal primo figlio di quel nodo ed infine dal terzo figlio.
- Questo ordinamento può essere utilizzato per attraversare in modo efficiente l'albero in ampiezza.



Come funziona quindi:

1) il nastro 1 contiene l'input w ed i nastri 2/3 sono vuoti
2) copia il nastro 1 sul nastro 2, inizializzando il nastro 3 a ε
3) usa il nastro 2 per simulare N con input "w" su un ramo di computazione. Prima di ogni passo di N, consulta il successivo sul nastro 3 e determina quale scelta fare. Se non rimangono simboli validi sul nastro 3 o la scelta non è valida si va al successivo passo anche se si rifiuta. Se si trova una configurazione accettante, accetta.
4) sostituire la stringa del nastro 3 con la stringa successiva nell'ordine delle stringhe. Simula il ramo successivo di N andando alla fase (2)

E dunque in conclusione:

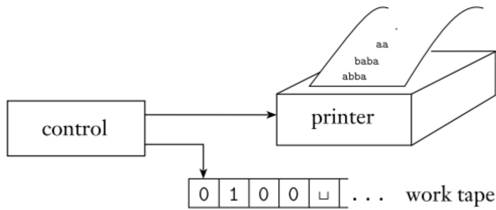
Corollary

Un linguaggio è Turing-riconoscibile se e solo se esiste una macchina di Turing non deterministica che lo riconosce.

$\Rightarrow$ Un linguaggio è Turing-riconoscibile se è riconosciuto da una TM deterministica, che è un caso particolare di TM non deterministica

$\Leftarrow$ Costruzione precedente

Una variante possibile (che citiamo e basta), sono gli *enumeratori*.



- **Enumeratore:** macchina di Turing + stampante
- Un enumeratore E inizia con nastro vuoto
- Di tanto in tanto, invia una striga alla stampante
- Linguaggio enumerato da E: tutte le stringhe stampate
- E può generare le stringhe in qualsiasi ordine, anche con ripetizioni

In un linguaggio L decidibile, si può costruire un enumeratore E (a prescindere dal suo input). Vengono date in input delle stringhe (con alfabeto finito e soprattutto tale che Σ* sia un insieme numerabile) in ordine crescente (lessicografico in questo caso) al decisore D e:

- se D accetta L, allora viene stampato l'input da E
- se D rifiuta L, allora non stampa nulla.

Anche da questa spiegazione, si può intuire che gli enumeratori sono Turing-equivalenti:

Theorem

Un linguaggio è Turing-riconoscibile se e solo se esiste un enumeratore che lo enumera

Idea: dobbiamo mostrare che

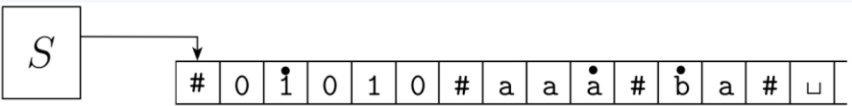
- se esiste un enumeratore E, allora esiste una TM M che riconosce lo stesso linguaggio
- se esiste una TM M che riconosce il linguaggio, allora possiamo costruire un enumeratore

Risposte Wooclap:

ESPERIMENTO CONCETTUALE:
Se cambiassimo la funzione di transizione di una TM in $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$, dove S significa "stai fermo", il nostro modello sarebbe più potente?

Risposta: No

Se nella macchina multinastro $\delta(q, 1, a, b) = (p, 0, a, a, L, R, R)$, quale sarà il contenuto del primo nastro virtuale dopo la simulazione della transizione?



Risposta:

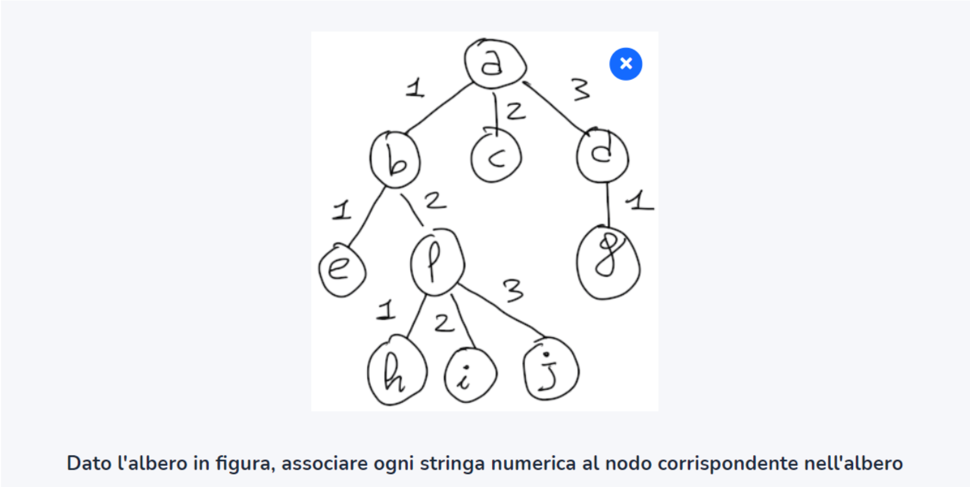
$\#\dot{0}0010\#$

Cosa succede sul secondo nastro virtuale?
Risposta

$\#aaa\dot{_}\#$

Se nella macchina multinastro $\delta(q, 1, a, b) = (p, 0, a, a, L, R, R)$, quale sarà il contenuto del terzo nastro virtuale dopo la simulazione della transizione?

Risposta: $\#a\dot{a}\#$



Dato l'albero in figura, associare ogni stringa numerica al nodo corrispondente nell'albero

ε – a
123 – i
12 – f
111-non esiste
3 - d

La TM deterministica che simula una TM non deterministica visita l'albero di computazione:
Risposta: In ampiezza

Quale dei nastri della TM deterministica D che simula una TM non deterministica N non viene mai scritto:
Risposta: Il primo, che tiene una copia nell'input

Cosa fa la TM deterministica D che simula una TM non deterministica N dopo aver sostituito la stringa sul nastro 3 con la stringa successiva rispetto alla vista in ampiezza dell'albero?
Risposta: Ricopia il nastro 1 sul nastro 2 e ricomincia la simulazione di N dall'inizio della computazione

In quale caso la TM deterministica D che simula una TM non deterministica N termina la computazione accettando l'input?
Risposta: appena si incontra una configurazione di rifiuto per N

In quale caso la TM deterministica D che simula una TM non deterministica N termina la computazione rifiutando l'input?
Risposta: se l'albero di computazione di N è infinito e non contiene configurazioni di accettazione

In quale caso la TM deterministica D che simula una TM non deterministica N non termina la computazione?
Risposte:

se tutti i rami dell'albero di computazione di N sono infiniti

se l'albero di computazione di N è infinito e non contiene configurazioni di accettazione

Una TM con "resta ferma" invece di "muovi a sinistra" ha una funzione di transizione:

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma\{S, R\}$$

Quale classi di linguaggi riconosce?

Risposta:
I linguaggi regolari

Algoritmi per macchine di Turing

Citiamo quindi David Hilbert, che l'8 agosto 1900 al Congresso Internazionale cita un "algoritmo" per determinare se un polinomio ha radice intera. Il presupposto era che l'algoritmo dovesse esistere ed era un problema non risolvibile algoritmicamente. La nozione di algoritmo esiste da sempre, ma l'idea è che senza una definizione formale, è impossibile provare che un algoritmo non esista.
Church pubblica un formalismo nel 1936 chiamato *λ-calcolo* per definire algoritmi; successivamente Turing pubblica le specifiche per una "macchina astratta" di definizione di algoritmi, dimostrando che certi modelli di computazione sono equivalenti. Il problema dell'algoritmo che stabilisce se un polinomio ha radici intere si è risolto nel 1970, stabilendo che non esiste.

Si parti quindi dal decimo problema di Hilbert, così descritto:

- Il decimo teorema di Hilbert con la nostra terminologia:

$$D = \{p \mid p \text{ è un polinomio avente radice intera}\}$$

E vogliamo capire quindi se *"D è un insieme decidibile?"*, mostrando che D sia Turing-riconoscibile, ma partiamo da un problema più semplice:

$$D_1 = \{p \mid p \text{ è un polinomio su } x \text{ avente radice intera}\}$$

La TM decide o riconosce il linguaggio $D = \{p \mid p$ è un polinomio su x con radice intera$\}$?

M_1 = "Su input $\langle p \rangle$, polinomio sulla variabile x:
1. Valuta p con x posta successivamente ai valori 0, 1, -1, 2, -2, ... Se in qualche momento la valutazione del polinomio è 0, accetta.

Esempio di polinomio: $3x^2 + 2x - 1 = 0$
Risposta:
Se il polinomio fosse a radici intere, accetta.
Se come in questo caso non fosse a radici intere, la macchina andrà avanti per sempre cercando di trovare un valore intero senza mai trovarlo, dunque andrà in loop e la macchina sarà un riconoscitore, non un decisore. Quindi:

Riconosce

Possiamo convertire questa macchina in un decisore perché possiamo calcolare i limiti entro i quali le radici di una singola variabile polinomiale devono stare e restringiamo la ricerca all'interno di questi limiti.
L'idea è di farlo stare tra questi valori:

$$\pm k \frac{c_{\max}}{c_1}$$

scritta dal prof come [-k * c_{max}/c_1 , +k * c_{max}/c_1] [-3   +3]
dove k è il numero di termini della polinomiale, c_{max} è il coefficiente con il valore più grande e c_1 è il coefficiente con il termine di ordine più alto. Se la radice non si trova entro questi limiti, la macchina rifiuta.
In ogni casi abbiamo a che fare con un problema *indecidibile*; potenzialmente, andando verso infinito, il problema potrebbe non avere mai alcuna soluzione come intuibile; è solamente riconoscibile (in termini di una TM).

Vogliamo quindi *descrivere una Turing Machine:*
- dando una *descrizione formale*, che dichiara esplicitamente tutto, dettagliatamente (esempio fatto nella prima lezione, ma diventa complicato farlo in casi meno banali ed è sconsigliato)
- dettagliando una *descrizione implementativa* del movimento delle testine e della scrittura su nastro a parole, senza dare il dettaglio sugli stati
- fornendo una *descrizione ad alto livello* fatta a parole dell'algoritmo senza dettagli implementativi (strada indicata da utilizzare sempre)

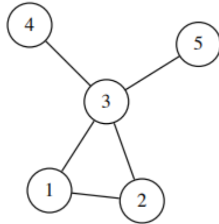
L'input è *sempre una stringa*; se avessimo degli oggetti diversi da una stringa, comunque deve essere rappresentabile come tale (es. polinomi, grammatiche, automi, ecc., avendo quindi l'input che può essere una combinazione diversa dei vari tipi di oggetti). In particolare:

- Un oggetto O codificato come stringa è $\langle O \rangle$.
- Una sequenza di oggetti $O_1, O_2, \ldots, O_k$ è codificata come $\langle O_1, O_2, \ldots, O_k \rangle$.
- L'algoritmo viene descritto con un testo, indentato e con struttura a blocchi.
- La prima riga dell'algoritmo descrive l'input macchina.

Immaginiamo di applicare l'idea al polinomio $3x^2 + 2x - 1 = 0$.
Descritto come stringa sarebbe:
3*x^2 + 2*x – 1
Σ = {0, …. , 9, + , - , * , ^}
In questo caso, con questa stringa enunciata, la macchina sarebbe nel formato corretto.
Se tuttavia avessimo una stringa come: 3 3xxy--* e vedendo quest'ultima rifiuta subito.

Una stringa come: 3*x*x + 2*x-1*x10
per rappresentare la stessa cosa andrebbe bene per un'altra macchina di Turing
ma per questa che esaminavamo no.

Riprendiamo quindi i *grafi*, usati estensivamente in informatica per migliaia di problemi computazionali, a noi utili per studiare alcuni problemi e dunque esaminarne la *classe di complessità*.
Formalmente, un *grafo* viene definito un insieme di *nodi* (o *vertici*) e da un'insieme di archi che collegano i nodi. A noi interessano in particolare i grafi *non orientati*, che semplicemente individuano una coppia di vertici indipendentemente dal loro ordine.



Definition (Grafo non orientato)

Un grafo non orientato (detto anche indiretto) G è una coppia (V, E) dove:

- $V = \{v_1, v_2, \ldots, v_n\}$ è un insieme finito e non vuoto di vertici;
- $E \subseteq \{\{u, v\} \mid u, v \in V\}$ è un insieme di coppie non ordinate, ognuna delle quali corrisponde ad un arco non orientato del grafo.

Un esempio di problema risolvibile tramite le TM per i grafi può essere:

- Un grafo è connesso se ogni nodo può essere raggiunto da ogni altro nodo tramite gli archi del grafo

Problema

Il linguaggio $A = \{\langle G \rangle \mid G \text{ è un grafo connesso}\}$ è decidibile?

Ad esempio, il grafo sopra è connesso. Possiamo descrivere questo problema come linguaggio, tale da crearci un decisore che accetta le stringhe che sono codifica di un grafo connesso.
Riportiamo una semplice descrizione ad alto livello:

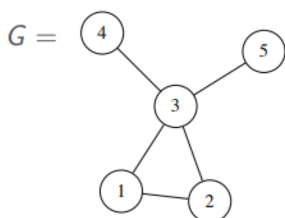
Descrizione di alto livello:

M = "Su input $\langle G \rangle$, la codifica di un grafo G:

1. Seleziona il primo nodo di G e lo marca.
2. Ripeti la fase seguente fino a quando non vengono marcati nuovi nodi:
3. per ogni nodo in G, marcalo se è connesso con un arco ad un nodo già marcato.
4. Esamina tutti i nodi di G: se sono tutti marcati, accetta, altrimenti rifiuta."

Si parte dal primo nodo che viene marcato e ciascuno viene identificato come numero decimale, connesso successivamente:

- Codifica di G: lista dei nodi + lista degli archi



$\langle G \rangle$ = (1,2,3,4,5)((1,2),(1,3),(2,3),(3,4),(3,5))

Questo esempio verifica che la macchina identifichi una lista partendo dagli archi, visti come coppie (liste di 2 elementi). Si controlla quindi per ogni operazione che tutti gli elementi siano distinti tra di loro (simile alla macchina di Turing vista settimana scorsa), verificando che anche gli archi siano tutti distinti tra loro rispetto ai vertici presenti nel grafo.

- M verifica che l'input sia sia una codifica di un grafo:
  - Se l'input non è nella forma corretta, rifiuta
  - Se l'input codifica un grafo, prosegue con la fase 1

Esaminiamo passo per passo il problema.
Il nastro all'inizio quindi è:

$\langle G \rangle = (1,2,3,4,5)((1,2),(1,3),(2,3),(3,4),(3,5))$

La fase 1 dice:

1. Seleziona il primo nodo di G e lo marca.

Dunque, diventa (per marcare al posto del puntino metto *, e sottolineo dove sono ora nel nastro):
(1*,2,3,4,5)((1,2),(1,3),(2,3),(3,4),(3,5))

La macchina avanza ad 1 (che sarà uguale al primo simbolo sottolineato) e poi su 2 (uguale al secondo simbolo sottolineato) nella parte destra dopo 1,2,3,4,5 quindi:
(1*,2*,3,4,5)((1,2),(1,3),(2,3),(3,4),(3,5))

Successivamente verifico anche (1,3) connessi, dunque:
(1*,2*,3,4,5)((1,2),(1,3),(2,3),(3,4),(3,5))

Vado dunque avanti con il terzo (2,3) e sottolineerò il quarto:
(1*,2*,3*,4,5)((1,2),(1,3),(2,3),(3,4),(3,5))

Poi abbiamo (3,4), dunque ho finito di esaminare anche 3 e 4 e li marco:
(1*,2*,3*,4*,5)((1,2),(1,3),(2,3),(3,4),(3,5))

Riparto dal primo e verifico (1,5), (2,5), ecc. fino a quando non trovo l'ultimo match con (3,5). Lo trovo rispetto a 3 appunto assieme al 5 tuttora sottolineato. Dunque, il nastro accettante è:
(1*,2*,3*,4*,5*)((1,2),(1,3),(2,3),(3,4),(3,5))

Concludiamo quindi dicendo che il problema in questo caso è chiaramente *decidibile*.

Domande Wooclap

Cosa è successo l'8 Agosto 1900?
Problemi di Hilbert: https://it.wikipedia.org/wiki/Problemi_di_Hilbert

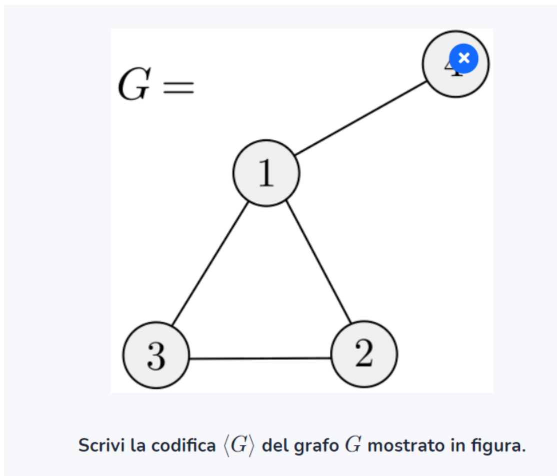
La TM decide o riconosce il linguaggio $D = \{p \mid p$ è un polinomio su x con radice intera$\}$?

M_1 = "Su input $\langle p \rangle$, polinomio sulla variabile x:
1. Valuta p con x posta successivamente ai valori 0, 1, -1, 2, -2, ... Se in qualche momento la valutazione del polinomio è 0, accetta.

Risposta: Riconosce

Cosa c'è che non va in questa macchina di Turing?
M_{bad} = "Su input $\langle p \rangle$, polinomio sulle variabili $x_1, \ldots, x_k$:
1. Per tutte possibili assegnazioni con valori interi di $x_1, \ldots, x_k$
2. Valuta p per ogni assegnazione
3. Se una di tali assegnazioni dà valore 0, accetta; altrimenti, rifiuta."

Risposta:
Questa macchina va sempre in loop in quanto continuerà a valutare il polinomio su tutte le possibili infinite combinazioni di input, memorizzandosi che ha trovato un certo valore e rimanendo lì finché non trova ciò che cerca. Questa TM è solo un riconoscitore, in quanto per come è scritta non accetterà mai.



Scrivi la codifica $\langle G \rangle$ del grafo G mostrato in figura.

- (1,2,3,4)((1,2),(1,3),(1,4),(2,3))

Il grafo H con codifica $\langle H \rangle = (1,2,3,4,5)((1,2),(2,3),(3,1)(4,5))$ è connesso?

Risposta: NO

Linguaggi decidibili

Vogliamo studiare una serie di problemi capendo se sono risolvibili o meno da parte di un algoritmo, dunque se siano o meno *decidibili*. Per questi linguaggi (sia per i regolari che per i CFL) vedremo:
- il problema dell'accettazione
- il test del vuoto
- il test sull'equivalenza

Cominciamo dal *problema dell'accettazione*:

- Problema dell'accettazione: testare se un DFA accetta una stringa

$$A_{DFA} = \{\langle B, w \rangle \mid B \text{ è un DFA che accetta la stringa } w\}$$

- B accetta w se e solo se $\langle B, w \rangle$ appartiene ad A_{DFA}
- Mostrare che il linguaggio è decidibile equivale a mostrare che il problema computazionale è decidibile

In questo caso il problema viene posto al livello di un DFA (quindi capire se A_{DFA} è decidibile), interpretato come se fosse una TM. La descrizione ad alto livello segue:

Idea: definire una TM che decide A_{DFA}

$M =$ "Su input $\langle B, w \rangle$, dove B è un DFA e w una stringa:
1. Simula B su input w
2. Se la simulazione termina in uno stato finale, accetta. Se termina in uno stato non finale, rifiuta."

La dimostrazione quindi considera una possibile codifica della stringa Q ed una per l'alfabeto di input:
<Q, Σ, δ, q_0, F> rappresentante la codifica della macchina M
$w_1,w_2, \dots w_k$ rappresentante la codifica della TM
Completamente:
<Q, Σ, δ, q_0, F, $w_1^*,w_2^*, \dots w_k^*$ >[stato corrente] → q_0 (*) = mark
Intendo inoltre memorizzare lo stato corrente in qualche modo, mettendo il pallino (mark) sul primo simbolo (q_0), scorrendo la funzione di transizione e trovando la coppia stato-simbolo sulla codifica, aggiornando lo stato corrente e spostando il pallino in avanti.
All'ultima transizione della simulazione:

- se lo stato è finale, accetta
- se lo stato non è finale, rifiuta

Passiamo poi all'idea di avere un NFA (quindi si vuole capire se A_{NFA} sia decidibile o meno). L'idea è di prendere l'ε-NFA, trasformarlo in un DFA tramite la costruzione a sottoinsiemi e fare in modo accetti la stringa "w". Poi, se essa accetta, il linguaggio è decidibile, altrimenti no.

$A_{NFA} = \{\langle B, w\rangle \mid B \text{ è un } \varepsilon\text{-NFA che accetta la stringa } w\}$

Più formalmente:

Dimostrazione:

$N =$ "Su input $\langle B, w\rangle$, dove B è un ε-NFA e w una stringa:

1. Trasforma B in un DFA equivalente C usando la costruzione per sottoinsiemi
2. Esegui M con input $\langle C, w\rangle$
3. Se M accetta, accetta; altrimenti, rifiuta."

N è un decisore per A_{NFA}, quindi A_{NFA} è decidibile

Vediamo lo stesso problema per una ER rappresentante un linguaggio:

$A_{REX} = \{\langle R, w\rangle \mid R \text{ è una espressione regolare che genera la stringa } w\}$

Molto similmente a prima, usiamo una TM che decide A_{NFA}, costruendo un ε-NFA equivalente e poi verificando che la TM sia un decisore.

Idea: usiamo la TM N che decide A_{NFA} come subroutine

Dimostrazione:

$P =$ "Su input $\langle R, w\rangle$, dove R è una espressione regolare e w una stringa:

1. Trasforma R in un ε-NFA equivalente C usando la procedura di conversione
2. Esegui N con input $\langle C, w\rangle$
3. Se N accetta, accetta; altrimenti, rifiuta."

P è un decisore per A_{REX}, quindi A_{REX} è decidibile

Ai fini della decidibilità, risulta equivalente dare in input alla TM in base ad un DFA, ε-NFA, ER, ecc.
La TM si pone come astrazione tale da poter convertire una codifica nell'altra.
Conseguentemente: *mostrare che un linguaggio sia decidibile equivale a mostrare che il problema stesso, computazionalmente, sia decidibile.*

Ora si ha un problema di diversa natura, dunque se un linguaggio di un certo automa sia vuoto oppure no (che significa sia ∅ ed E, nella definizione del linguaggio, rappresenta la parola *emptiness*).
Tale problema è definito come *test del vuoto* e si vuole esaminarlo partendo da un DFA:

- Negli esempi precedenti dovevamo decidere se una stringa appartenesse o no ad un linguaggio
- Ora vogliamo determinare se un automa finito accetta una qualche stringa

$$E_{DFA} = \{\langle A\rangle \mid A \text{ è un DFA e } L(A) = \emptyset\}$$

- Puoi descrivere un algoritmo per eseguire questo test?

La dimostrazione formale è verificare se esiste uno stato finale che può essere raggiunto da quello iniziale.

T = "Su input $\langle A\rangle$, la codifica di un DFA A:

1. Marca lo stato iniziale di A.
2. Ripeti la fase seguente fino a quando non vengono marcati nuovi stati:
3. marca ogni stato di A che ha una transizione proveniente da uno stato già marcato.
4. Se nessuno degli stati finali è marcato, accetta; altrimenti rifiuta."

Un altro classico problema è *il test di equivalenza*:

$$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ e } B \text{ sono DFA e } L(A) = L(B)\}$$

L'idea è:

- costruiamo un DFA C che accetta solo le stringhe che sono accettate da A o da B, ma non da entrambi
- se $L(A) = L(B)$ allora C non accetterà nulla
- il linguaggio di C è la differenza simmetrica di A e B

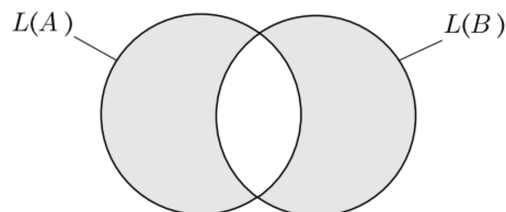
Quindi a livello dimostrativo/formale:

- la differenza simmetrica di A e B è:

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

- i linguaggi regolari sono chiusi per unione, intersezione e complementazione
- F = "Su input $\langle A, B\rangle$, dove A e B sono DFA:
  1. Costruisci il DFA C per differenza simmetrica
  2. Esegui T, la TM che decide E_{DFA} con input $\langle C\rangle$
  3. Se T accetta, accetta; altrimenti, rifiuta."

A livello insiemistico è visibile in questo modo:



Ora vediamo i *problemi per linguaggi Context-Free*.
Quindi abbiamo il *problema dell'accettazione*, costruendo anche qui un decisore:

$$A_{CFG} = \{\langle G, w\rangle \mid G \text{ è una CFG che genera la stringa } w\}$$

Idea: costruiamo una TM che provi tutte le derivazioni di G per trovarne una che genera w

La *strategia non funziona*, infatti si va avanti all'infinito (grammatica con infinite derivazioni, dato che si può letteralmente mettere qualsiasi stringa); come tale non si può costruire un decisore.
Se la CFG è in FNC/Forma Normale di Chomsky, allora ogni derivazione ha esattamente *2|W|-1 passi (es. 14 nella parte 1 del file di preparazione all'esame, soluzione presente su Mega)*. Ciò è utile perché permette di riconoscere esattamente un linguaggio regolare da uno che non lo è.
La dimostrazione per A_{CFG} segue questa idea:

S = "Su input $\langle G, w\rangle$, dove G è una CFG e w una stringa:

1. Converti G in forma normale di Chomsky
2. Elenca tutte le derivazioni di $2|w| - 1$ passi. Se $|w| = 0$, elenca tutte le derivazioni di lunghezza 1
3. Se una delle derivazioni genera w, accetta; altrimenti rifiuta."

Esaminiamo ora di nuovo il *test del vuoto*, nel caso ora del linguaggio CF:

$$E_{CFG} = \{\langle G\rangle \mid A \text{ è una CFG ed } L(G) = \emptyset\}$$

Bisogna quindi procedere in un altro modo (non usando la S di prima), ma per esempio capendo se ogni variabile sia in grado di generare una stringa di terminali.

In maniera estesa:

R = "Su input $\langle G\rangle$, la codifica di una CFG G:

1. Marca tutti i simboli terminali di G.
2. Ripeti la fase seguente fino a quando non vengono marcate nuove variabili:
3. marca ogni variabile A tale che esiste una regola $A \to U_1 \dots U_k$ dove ogni simbolo $U_1 \dots U_k$ è già stato marcato.
4. Se la variabile iniziale non è marcata, accetta; altrimenti rifiuta."

Per esempio vediamo la grammatica:
S → SB
B → Ba*
B*→ ε*

La variabile iniziale non è stata marcata e quindi accetta; se invece venisse marcata allora rifiuta correttamente (questo perché marco una variabile e poi va in loop).
Concretamente:
S → SB → S → SB → ....

Da qui poi prendiamo il caso della *differenza simmetrica*. Questo è un caso interessante, in quanto è appunto una operazione di differenza. Essendo dei CFG saranno uguali fintanto che accettano le stesse stringhe; quindi, sono definibili da un unico automa, ad esempio. L'idea quindi è di utilizzare la stessa tecnica di EQ_{DFA} (quindi costruire un DFA che accetta solo le stringhe di A o di B; se sono uguali non accetta nulla).

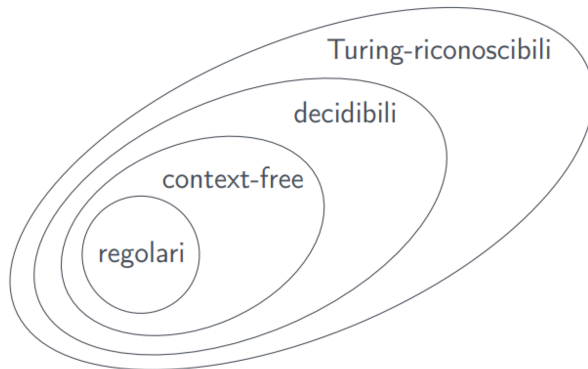
$$EQ_{CFG} = \{\langle G, H\rangle \mid G \text{ e } H \text{ sono CFG e } L(G) = L(H)\}$$

A queste condizioni si calcola la differenza simmetrica tra G ed H per provare l'equivalenza.
L'unico problema è che l'idea funziona, ma non nel contesto delle grammatiche context-free.
Esse infatti non sono chiuse per complementazione/intersezione e semplicemente non accetteranno un'operazione come la differenza. Si conclude che EQ_{CFG} non sia decidibile.
In aggiunta a questo si deve fare una considerazione: la CFG potrebbe avere dei rami di computazione infinita, ricorsivamente, pertanto potrebbe non terminare mai.
Quindi, ad alto livello, l'idea è di usare una TM M che simula <G, H> e le rispettive derivazioni sul nastro.
Dato che l'intersezione è un'operazione non possibile, potenzialmente, la macchina andrà in loop cercando di marcare, partendo da una delle due CFG o su entrambe, i simboli terminali. In alcuni casi può anche essere decidibile, ma formalmente è solo Turing-riconoscibile.

Scritto da Gabriel

Con una pila è facile simulare una TM, inoltre sappiamo che le TM nondeterministiche possono essere simulate, anche passando per mezzo di automi, da TM deterministiche. In merito alle pile si deve sempre considerare l'equivalenza che sussiste tra le stesse e le CFG. Ciò permette di risolvere questi esercizi (quindi: una pila che accetta/rifiuta lo stato X nello stack corrisponde ad una grammatica che attua una particolare derivazione).
Dunque, ogni CFL è decidibile; basterà costruire una TM decisore che decide A_{CFG} per il linguaggio L. Semplicemente esegue la TM S con input <G, w> e se S accetta, allora *accetta*, altrimenti *rifiuta*.

Queste dunque le relazioni tra i linguaggi, intese anche come *classi di capacità computazionale.*



<u>*Risposte Wooclap*</u>

Dato il DFA M in figura, indica quali delle seguenti stringhe appartengono al linguaggio A_{DFA} (puoi scegliere più di una risposta)

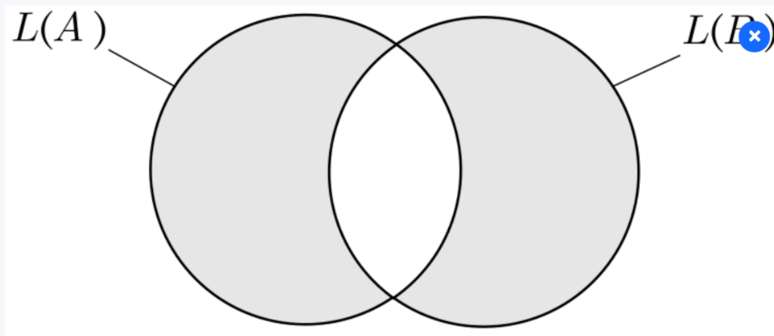
Risposte: <M , 0100> <M, 000>

Questa TM è un riconoscitore o un decisore per E_{DFA}?
T' = "Su input $\langle A \rangle$, dove A è un DFA:
1. Per ogni stringa $w \in \Sigma^*$, seguendo l'ordine lessicografico:
2. Esegue la TM M che decide A_{DFA} su $\langle B, w \rangle$. Se M accetta, RIFIUTA
3. Se nessuna stringa viene accettata, ACCETTA"

Risposta: Essendo una macchina che va in loop quando dovrebbe accettare e accetta quando dovrebbe fermarsi. Dunque, non può essere certo un decisore, ma neanche un riconoscitore.
Riconosce quindi un linguaggio diverso.

E quindi: nessuna delle precedenti

Scritto da Gabriel



Nella figura, la parte in grigio è la *differenza simmetrica* di $L(A)$ e $L(B)$. Cosa succede alla differenza simmetrica quando $L(A) = L(B)$?

Risposta: L'insieme vuoto - ∅

Considera i linguaggi context-free $A = \{a^n b^n c^m \mid m, n \geq 0\}$ e $A = \{a^m b^n c^n \mid m, n \geq 0\}$. Quale linguaggio ottieni se fai l'INTERSEZIONE di A e B?

Risposta: $a^n b^n c^n$, n >= 0

**Sia $A_{\varepsilon CFG} = \{\langle G \rangle \mid G$ è una CFG che genera $\varepsilon\}$.
Completa la descrizione della seguente TM che decide $A_{\varepsilon CFG}$:**

Risposta:
V = Su input <G>, dove G è una CFG:
1) Esegue la TM che decide A_CFG su input <G, ε>
2) Se questa macchina accetta, *accetta* altrimenti *rifiuta*

Indecidibilità

Esiste un problema specifico che è algoritmicamente irrisolvibile, dunque problemi di interesse pratico e non solo teorico. Un esempio concreto è la *verifica del software*, dunque il processo che considera un generico programma e verifica che, per lo scopo implementativo del suo algoritmo, si può definire corretto. Nessun algoritmo potrà mai fornire un esempio di correttezza universale per un software.
Prendiamo un primo esempio di problema indecidibile, prendendo il seguente problema:

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ è una TM che accetta la stringa } w\}$$

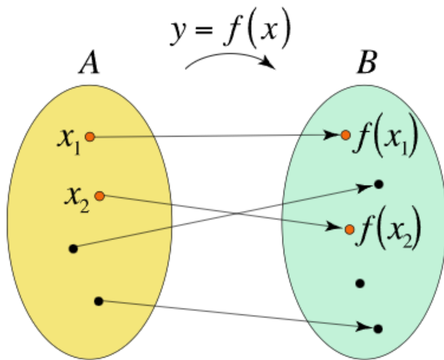
Questo linguaggio è Turing-riconoscibile (si può dimostrare operando per simulazione). Si vede quindi che i riconoscitori sono più potenti dei decisori. Semplicemente:

- U = "Su input $\langle M, w \rangle$, dove M è una TM e w una stringa:
  1. Simula M su input w
  2. Se la simulazione raggiunge lo stato di accettazione, accetta; se raggiunge lo stato di rifiuto, rifiuta."

Per dimostrare che il linguaggio non è decidibile (e dunque U non è riconoscitore), dobbiamo fare alcune osservazioni. La macchina *U* è un esempio di *Macchina Universale di Turing*, dato che è in grado di simulare qualsiasi macchina di Turing partendo dalla sua descrizione.

La dimostrazione che afferma A_{TM} usa la *diagonalizzazione*, metodo che serve a confrontare dimensioni di insiemi infiniti, non potendo contare gli insiemi in forma finita (es. diagrammi di Venn).
L'idea è di avere due insiemi finiti con stessa dimensione e gli elementi di uno possono essere accoppiati con gli elementi dell'altro.

Formalmente voglio trovare una funzione *f* tale da far corrispondere elementi diversi di A ad elementi diversi di B (*iniettiva*) toccando tutti gli elementi diversi (*suriettiva*).



Partendo quindi da due insiemi A e B ed una funzione f : A → B (considerando che hanno la stessa cardinalità se esiste una funzione biettiva, quindi sia suriettiva/iniettiva):

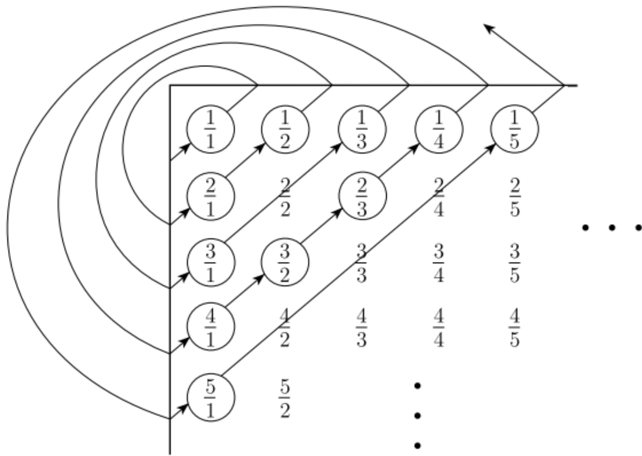
- f è iniettiva se non mappa mai elementi diversi nello stesso punto: $f(a) \neq f(b)$ ogniqualvolta che $a \neq b$
- f è suriettiva se tocca ogni elemento di B: per ogni $b \in B$ esiste $a \in A$ tale che $f(a) = b$
- Una funzione iniettiva e suriettiva è chiamata biettiva: è un modo per accoppiare elementi di A con elementi di B

Passiamo ad alcuni esempi concreti, discutendo in particolare il problema degli insiemi numerabili, sapendo che un insieme viene definito come tale se è finito oppure ha la stessa cardinalità (cioè numero di elementi) di **N**. (le domande sono duplicate anche in Wooclap, per questo presenti due volte):

- Qual è il più grande tra l'insieme dei numeri naturali **N** e l'insieme dei numeri pari?
  Risposta: Hanno la stessa dimensione
  Per poterlo dimostrare, l'idea è di prendere k moltiplicandolo per 2, ottenendo una funzione biiettiva (ogni numero pari ha un'associazione e riguarda tutti i numeri pari).
  Matematicamente:
  $f(x) = 2K$
  0 0
  1 2
  2 4
  … …
  Infatti, la cardinalità non considera il tipo di elementi o il loro ordine; ciò considera, inoltre, che se c'è un'iniezione da un insieme A ad un insieme B, non vale il contrario. Questa è la chiave per capire, almeno a livello di cardinalità, come ragionare capendo chi è il più grande o il più piccolo.
  A livello concreto, hanno stessa dimensione perché un tipo di funzione suriettiva associa esattamente lo stesso numero di elementi.

- L'insieme dei razionali **Q** è numerabile?
  Definiamo **Q** = {m/n | m,n ∈ **N**). L'idea iniziale potrebbe essere quella di listare gli elementi fino ad N, accoppiando idealmente ogni elemento di **Q** all'elemento corrispondente in **N**. Il problema è uno: gli elementi sono infinito, pertanto non è un approccio buono da seguire.
  Invece di listare ogni singolo elemento per righe, listiamo gli elementi della diagonale, sovrapposto nel diagramma, iniziando dall'angolo. I primi elementi della diagonale contengono solo l'elemento

1/1, la seconda 2/1, 1/2. Nella terza diagonale, si ha una complicazione, poiché contiene 3/1, 2/2, 1/3. Aggiungendoli alla lista ripeteremmo 1/1=2/2, pertanto saltiamo elementi come questo dato che possono essere ripetuti. Seguendo questo principio, listiamo ogni elemento di **Q**.
Concludendo, essendo comunque insiemi infiniti, non ci sarà mai corrispondenza per ogni elemento di **N**; pertanto, il sistema non può dirsi numerabile.



- L'insieme dei numeri reali **R** è numerabile?
Dobbiamo dimostrare che non ci sono corrispondenze tra i reali ed i naturali e la prova è per assurdo. Creiamo una funzione *f* che faccia corrispondere tutti gli elementi di **N** con tutti gli elementi di **R**. Si cerca quindi di costruire concretamente facendo esempi numerici, affinché ad un punto corrisponda un valore come segue:

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
⋮	⋮

Ogni numero è rappresentato in forma decimale, tenendo le cifre significative oltre il punto decimale. L'idea che vorremmo è che x ≠ f(n) per ogni n. Per assicurare ad esempio che x ≠ f(1), facciamo in modo la prima cifra di x sia qualsiasi cifra diversa dal primo decimale di frazione, similmente per avere x ≠ f(2), facciamo in modo la seconda cifra di x sia qualsiasi cifra diversa dal secondo decimale di frazione, così via. In questo modo, si procede in diagonale.
Così capiamo che *x* non sarà f(n) per ogni n, in quanto diversa nella n-esima cifra frazionaria.
L'idea grafica comunque è:

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
⋮	⋮

$x = 0.4641\ \ldots$

Questo esempio dimostra che alcuni linguaggi non sono decidibili oppure Turing-riconoscibili, perché ci sono un numero infinito di linguaggi ma solo un insieme contabile di macchine di Turing. Pertanto, *alcuni linguaggi possono non essere Turing-riconoscibili.*

- Dato un alfabeto finito Σ, Σ* è numerabile?
L'insieme di stringhe è contabile per ogni alfabeto, dato che avendo solo un certo numero di stringhe finite per ciascuna lunghezza, potremmo semplicemente scrivere tutte le stringhe.
Osserviamo prima di tutto che l'insieme di tutte le sequenze binarie finite non è contabile. Una *sequenza binaria infinita* è tale per tutti gli 0 ed 1. Avendo B insieme di tutte le sequenze binarie infinite, usiamo anche qui una prova di diagonalizzazione.

Partendo da L insieme di tutti i linguaggi sull'alfabeto Σ, cerchiamo di dimostrare non sia numerabile dando una corrispondenza a B, facendo in modo siano della stessa dimensione. Ciascun linguaggio A ha un'unica sequenza in B. L'i-esimo bit della sequenza è un 1 se $s_1 \in A$ ed uno 0 se $s_1 \notin A$, definita come *sequenza caratteristica* di A.
Per esempio, se A fosse il linguaggio di tutte le stringhe che iniziano con 0 sull'alfabeto {0,1}, la sequenza caratteristica X_A sarebbe:

$$\begin{array}{lcccccccccc} \Sigma^* = \{ & \varepsilon, & 0, & 1, & 00, & 01, & 10, & 11, & 000, & 001, & \cdots \} ; \\ A = \{ & & 0, & & 00, & 01, & & & 000, & 001, & \cdots \} ; \\ \chi_A = & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \cdots . \end{array}$$

Dato che non ci sarebbe corrispondenza con tutti gli elementi, allora L è non numerabile.

- L'insieme di tutte le macchine di Turing è numerabile?
  Esso è numerabile, in quanto nonostante esistano infinite stringhe, almeno una di questa ha lunghezza finita. Inoltre, c'è un numero finito di stringhe per quella lunghezza. Perciò questo insieme contiene un infinito numero di elementi di lunghezza finita.

- L'insieme di tutte le sequenze binarie infinite è numerabile?
  L'idea è di procedere per assurdo. Supponendo S sia numerabile e non un insieme finito, naturalmente, dato che esiste una infinita sequenza di simboli che contiene ogni elemento di S almeno una volta. Otterremo una contraddizione esibendo un elemento dell'insieme tale che, come visto prima, non appartenga a f_n. Per ogni intero positivo *i*, diamo una f_i definita su tutti i valori. Per ogni intero positivo, l'ennesimo elemento è diverso da $b_{n,n}$, n-esimo elemento di f_n. Questo giustifica la contraddizione e l'insieme è non numerabile.

- Dato un alfabeto finito Σ, l'insieme di tutti i linguaggi su Σ* è numerabile?
  Vediamo l'esempio su Σ = {a,b}. Possiamo associare ad ogni simbolo una corrispondente stringa. Ad ogni stringa aggiungiamo un 1 di fronte ad ogni stringa e interpretiamo tutti questi numeri come binari ed L è numerabile, avendo poi che l'idea si estende a L_i. Dato che ogni i-esimo insieme è numerabile, possiamo costruire una tabella con gli indici di colonna che sono gli indici del linguaggio attuale e quelli di riga che indicano se contiene la stringa s_j, quindi nel nostro esempio:

aa	ε	a	b	aa	ab	ba	bb	aaa	…
0	0	0	0	0	0	0	0	0	
L_1	0	1	0	1	1	0	0	1	
…	….	….	….	….	….	….	….	….	
…	….	….	….	….	….	….	….	….	

Ora invertiamo il valore di ogni elemento nelle celle della diagonale, collezionando poi tutte le stringhe s_j affinché la cella diagonale della colonna s_j abbia un 1 dopo:

	s_1	s_2	s_3	s_4	…
L_1	**0**	0	1	1	…
L_2	0	**1**	0	1	…
L_3	1	1	**1**	0	…
L_4	0	0	1	**0**	…
⋮	⋮	⋮	⋮	⋮	⋱

Chiamandola $L_{diag} = s_2, s_3$, ….
L_{diag} è diverso per ogni linguaggio L_i. Questo implica che $L_{diag} \neq L_i$ per ogni L_i e pertanto non è numerabile.

L'insieme di tutte le macchine di Turing è numerabile, mentre l'insieme di tutti i linguaggi è non numerabile. Sappiamo che *devono* esistere linguaggi non riconoscibili da una macchina di Turing.
Tornando al problema iniziale:

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ è una TM che accetta la stringa } w\}$$

Ragioniamo per contraddizione. Supponendo A_{TM} decidibile immaginiamo ci sia un H decisore.
Cosa fa H sull'input <M, w>?

$$H(\langle M, w\rangle) = \begin{cases} \text{accetta} & \text{se } M \text{ accetta } w \\ \text{rifiuta} & \text{se } M \text{ non accetta } w \end{cases}$$

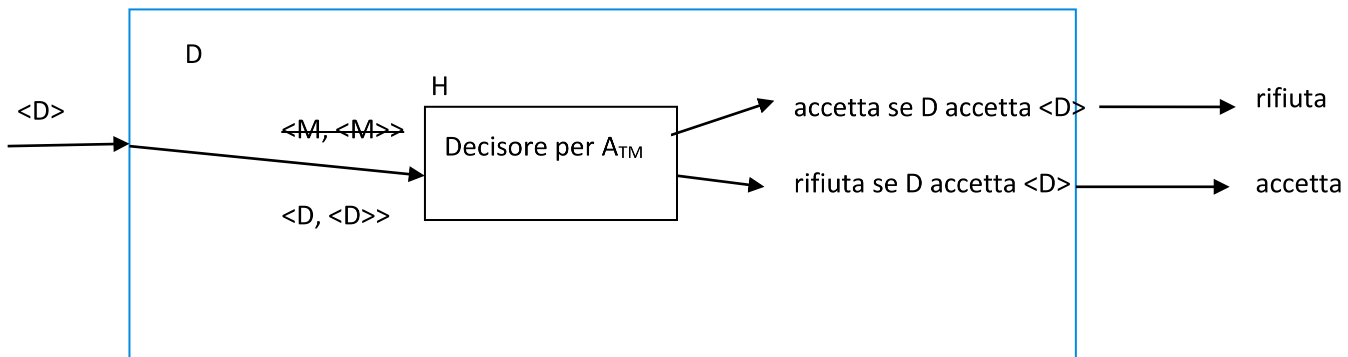
Immaginiamo di definire D come subroutine e, come tale:

- esegue H sull'input <M, <M>>
- da in output l'opposto di quanto mandi H. Perciò, se H accetta, *rifiuta*. Se H rifiuta, allora *accetta*.

Definito come appunto programma, l'idea in compilazione sarebbe la seguente:

$$D(\langle D\rangle) = \begin{cases} \text{accetta} & \text{se } D \text{ non accetta } \langle D\rangle \\ \text{rifiuta} & \text{se } D \text{ accetta } \langle D\rangle \end{cases}$$

Dato che D sarebbe forzato ad eseguire l'opposto, pertanto sarebbe evidentemente una contraddizione.
Graficamente:



Riassumendo e commentando i principi della dimostrazione:

1. H accetta $\langle M, w\rangle$ esattamente quando M accetta w
   a. Banale: abbiamo assunto che H esista e decida A_{TM}
   b. M rappresenta qualsiasi TM e w è una qualsiasi stringa
2. D rifiuta $\langle M\rangle$ esattamente quando M accetta $\langle M\rangle$
   a. Cosa è successo a w?
   b. w è solo una stringa, come $\langle M\rangle$. Tutto ciò che stiamo facendo è definire quale stringa dare in input alla macchina.
3. D rifiuta $\langle D\rangle$ esattamente quando D accetta $\langle D\rangle$
   a. Questa è la contraddizione.

Dove si usa quindi la diagonalizzazione? Diventa evidente quando si esaminano le tabelle di comportamento per le TM H e D. In queste tabelle listiamo tutte le TM lungo le righe (quindi M_1, M_2, ecc.) e le loro descrizioni (quindi stringhe che accettano lungo le colonne (<M_1>, <M_2>, … <M_n>) lungo le colonne.
Questi dati determinano se la macchina in una certa riga accetta l'input di una certa colonna. La entry sarà *accept* se la macchina accetta l'input ma è *blank* se rifiuta o va in loop su un certo input.

Graficamente:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

Continuando, le entry sarebbero il risultato di esecuzione di H sugli input corrispondenti alla precedente. Aggiungendo D alla figura, per nostro assunto, H è una TM e altrettanto è D. Perciò deve essere presente nella lista M_1, M_2, … di tutte le TM. Si noti che D computa l'opposto delle entry diagonali. La contraddizione si ha nella casella con il punto interrogativo, dato che la entry deve essere l'opposto in realtà:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$			$\vdots$				$\ddots$

Domande Wooclap

Considera i seguenti insiemi: quale dei due è più grande?
$\mathbb{N} = \{0, 1, 2, \ldots, \}$, **insieme dei numeri naturali**
$\mathbb{E} = \{0, 2, 4, \ldots, \}$, **insieme dei numeri pari**

Risposta: Hanno la stessa dimensione

L'insieme $\mathbb{Q} = \{\frac{m}{n} \mid m, n \in \mathbb{N}\}$ dei numeri razionali positivi è numerabile?

Risposta: Non è numerabile

L'insieme $\mathbb{R}$ dei numeri reali è numerabile?

Risposta: Non è numerabile

Dato un alfabeto finito Σ, Σ^* è numerabile?

Risposta: Non è numerabile

L'insieme di tutte le macchine di Turing è numerabile?

Risposta: É numerabile

L'insieme di tutte le sequenze binarie infinite è numerabile?

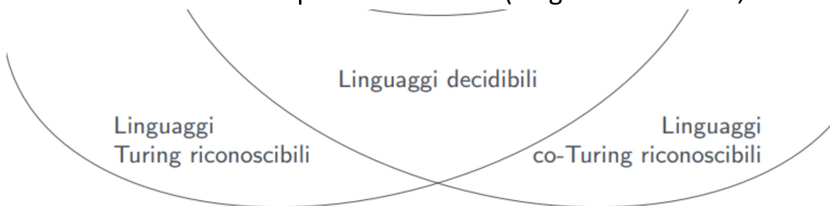
Risposta: Non è numerabile

Dato un alfabeto finito Σ, l'insieme di tutti i linguaggi su Σ^* è numerabile?

Risposta: Non è numerabile

Non decidibilità e Riducibilità

Abbiamo quindi visto che A_{TM} è Turing-riconoscibile e, sapendo che l'insieme di tutte le TM è numerabile, l'insieme di tutti i linguaggi è non numerabile. Conseguentemente, deve esistere un linguaggio non Turing-riconoscibile. Prima di poter mostrare un linguaggio non Turing-riconoscibile, vogliamo mostrare che se un linguaggio ed il suo complementare sono Turing-riconoscibili, allora il linguaggio è *decidibile*.
Il linguaggio è *co-Turing riconoscibile* se il complementare di un linguaggio è Turing-riconoscibile (e quindi è un complemento di un linguaggio anch'esso Turing-riconoscibile).
Graficamente abbiamo questa situazione (diagrammi di Venn, cioè diagrammi insiemistici):



Il teorema che lo completa è:

Un linguaggio è decidibile se solo se è Turing-riconoscibile e co-Turing riconoscibile.

Vogliamo dimostrare sia la direzione di A che di $\underline{A}$ e quindi:

- Se A è decidibile, allora sia A che $\overline{A}$ sono Turing-riconoscibili
  - Il complementare di un linguaggio decidibile è decidibile!
- Se A e $\overline{A}$ sono Turing-riconoscibili, possiamo costruire un decisore per A

L'idea è di avere una macchina M decisore, ma diciamo che M_1 è riconoscitore per A ed M_2 è riconoscitore per $\underline{A}$. Vogliamo creare quindi M tale che sia un decisore e:
M_1 riconoscitore per A
M_2 riconoscitore per $\underline{A}$
Si avvia la simulazione con M su input w

1) Eseguo M_1 con input w, se accetta allora *accetta*
2) Eseguo M_2 con input w, se accetta allora *rifiuta*

Il problema è che M_1, potenzialmente, potrebbe andare in loop (non riuscirebbe ad accettare tutte le stringhe e si pianta), non ottenendo un decisore.
Al posto di eseguirne prima una e poi l'altra, provo ad eseguirle in parallelo.

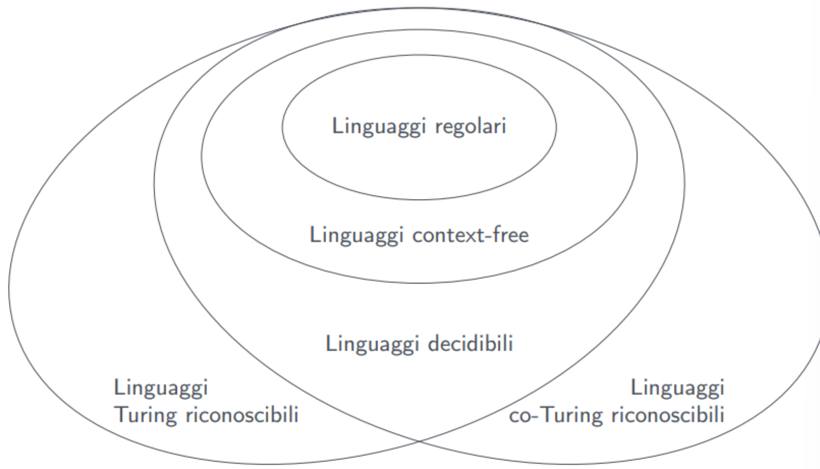
Ripeti quanto segue:

1) esegue un passo di computazione di M_1 su W. Se accetta, *accetta*
2) esegue un passo di computazione di M_2 su W. Se accetta, *rifiuta*

Ripeti da (1). Eseguendo le due macchine in parallelo, significa che M ha due nastri (uno per M_1 ed un altro per M_2). Dato che ogni stringa w sta in A oppure $\underline{A}$, allora, ad un certo punto, una delle due macchine accetterà. Siccome M si ferma in qualsiasi caso si fermi M_1 oppure M_2, allora si ferma sempre e la macchina M sarà un decisore. Se esiste un loop in M_1 vuol dire che sarà nel complementare e, se ci accorgiamo che una delle due rifiuta, mi fermo e so che esiste un decisore.

- Se il complementare di A_{TM} fosse Turing-riconoscibile, allora A_{TM} sarebbe decidibile
- Sappiamo che A_{TM} non è decidibile, quindi il suo complementare non può essere Turing-riconoscibile!

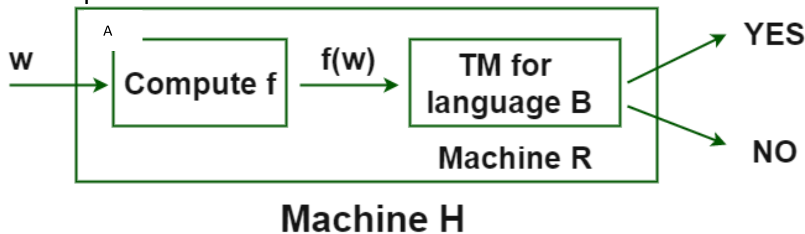
Concludendo, graficamente:



Affrontiamo il *problema della fermata* (*halting problem*):

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ è una TM che si ferma su input } w\}$$

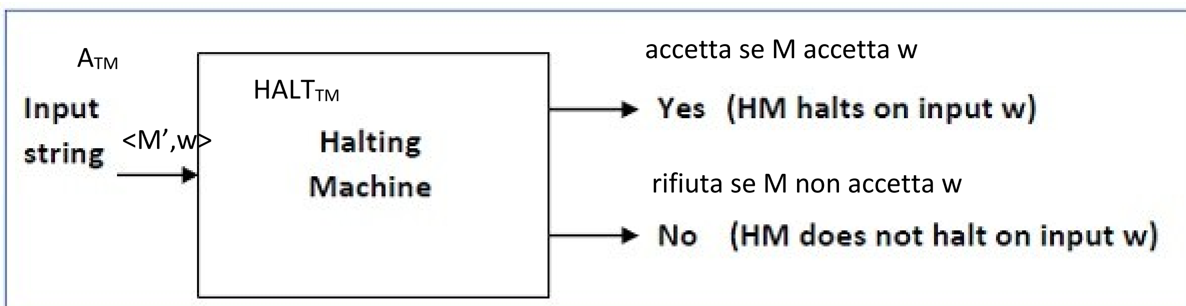
Non possiamo dimostrarne l'indecidibilità tramite diagonalizzazione, ma A_{TM} è indecidibile; vogliamo sfruttare questo fatto per operare la dimostrazione.
La prova è per assurdo, assumendo che $HALT_{TM}$ sia decidibile e usando l'assunzione che A_{TM} sia decidibile.
A questo punto si discute la ***riduzione***, per poter trasformare un problema in un altro, tale che *la soluzione del secondo problema serva a risolvere il primo*.
L'esempio della riduzione di A in B:



Diciamo anche che:

- se A è riducibile a B e B è decidibile, allora A è decidibile
- se A è riducibile a B e A è indecidibile, allora B è indecidibile

Quindi, costruendo una TM R che risolve A_{TM} si usa una TM che prende $HALT_{TM}$ come subroutine.



Usiamo R per costruire una macchina decisore S, che accetta sulla base dell'input <M, w> se M accetta e rifiuta se M va in loop oppure rifiuta w. In questo caso, però, non sempre si potrebbe determinare la fine del loop e la simulazione non terminerà e l'idea così descritta non funziona.
Invece, si usa il presupposto di avere la TM R che decide $HALT_{TM}$. Con R si può verificare se M si ferma su w. Se R indica che M non si ferma su w, rifiuta perché <M, w> non è in A_{TM}. Tuttavia, se R indica che M si ferma su w, si può fare la simulazione senza alcun pericolo di looping.

Quindi, se la TM R esiste, possiamo decidere A_{TM}, ma sappiamo che A_{TM} è indecidibile. In virtù di questa contraddizione, possiamo concludere che R non esiste. Pertanto, $HALT_{TM}$ è indecidibile.
Per completare la dimostrazione, assumiamo allo scopo di ottenere una contraddizione che la TM R decida $HALT_{TM}$. Costruiamo una TM S che decide A_{TM} ed opera come segue:
S = "Sull'input <M, w> una codifica con una TM M e una stringa w:
1) Si esegue la TM R sull'input <M,w>
2) Se R rifiuta, *rifiuta*.
3) Se R accetta, si simula M su w fino a quando non si ferma.
4) Se M ha accettato, *accetta*; se M ha rifiutato, *rifiuta*."
L'idea è che l'input si fermi sempre e verifica attraverso un "trucco" che la macchina di riferimento abbia l'input a lei appartenente; se lo fa, esegue M su w e accetta quando lei lo fa, altrimenti rifiuta.
La macchina idealmente si ferma su ogni singolo input, perché:

- se la stringa appartiene al linguaggio, allora viene eseguita la TM su w e accetta.
  Quindi f<<M,w> = $\langle M_w \rangle \in HALT_{TM}$
- se la stringa non appartiene al linguaggio, allora viene eseguita la TM su w ma la computazione va in loop o rifiuta.
  Quindi f<<M,w> = $\langle M_w \rangle \in HALT_{TM}$

Formalmente, la macchina comunque si ferma a prescindere, sia dando un input che il suo opposto. Lei non prende in input una stringa, ma il problema. La riduzione ha senso, si ferma in ogni caso e la macchina M fa quello che le viene chiesto. Abbiamo dimostrato che $A_{TM} \leq_m HALT_{TM}$, ma A_{TM} è indecidibile e allora anche $HALT_{TM}$ è indecidibile.

Normalmente per capire se un problema è indecidibile, si usa questa traccia, per le ***dimostrazioni per riduzione***:

1. Assumi che B sia decidibile
2. Riduci A al problema B
   - costruisci una TM che usa B per risolvere A
3. Se A è indecidibile, allora questa è una contraddizione
4. L'assunzione è sbagliata e B è indecidibile

Vediamo l'esempio del *problema del vuoto:*

$$E_{TM} = \{\langle M \rangle \mid M \text{ è una TM tale che } L(M) = \emptyset\}$$

- La dimostrazione è per contraddizione e riduzione di A_{TM}
- Chiamiamo R la TM che decide E_{TM}
- Useremo R per costruire la TM S che decide A_{TM}

Seguiamo lo stesso pattern visto adesso. Assumiamo che E_{TM} sia decidibile e poi mostriamo che lo sia, ragionando per contraddizione. Poniamo R una TM che decide E_{TM}. Usiamo R per costruire una TM S che decide A_{TM}. Un'idea sarebbe quelle di eseguire R sull'input <M> in S e vedere se accetta.
Se R rifiuta <M>, sappiamo solo che L<M> non è vuoto, perciò che M accetta qualche stringa, ma non abbiamo alcuna informazione in merito alla stringa "w".
Invece di eseguire R su <M>, eseguiamo R su una modifica di <M>, cioè garantiamo che M rifiuti tutte le stringhe eccetto "w", ma sull'input di "w" lavora come al solito. Poi usiamo R per determinare se la macchina modificata riconosce il linguaggio vuoto. L'unica stringa che la macchina può ora accettare è "w"; perciò, il suo linguaggio sarà non vuoto se accetta "w". Se R accetta la descrizione della macchina modificata, sappiamo che la macchina modificata non accetta nulla e che M non accetta "w".

Per completare la prova, descriviamo la macchina modificata come M_1. La descrizione ad alto livello è:
M_1 = Sull'input x:

1) se x ≠ w, *rifiuta*
2) se x = w, esegui M sull'input "w" e *accetta* se M lo fa.

Questa macchina ha la stringa "w" come parte della sua descrizione. Conduce il test su *x = w* nel modo classico, scansionando l'input e confrontandolo carattere per carattere con "w" per determinare se sono uguali. Mettendo insieme tutto questo, assumiamo che la TM R decida E_{TM} e costruiamo la TM S che decide A_{TM} come segue.
S = Sull'input <M, w>, una codifica della TM M ed una stringa w:

1) Si usa la descrizione di M e w per costruire la TM M_1 appena descritta
2) Si esegue R sull'input <M_1>
3) Se R accetta, *rifiuta*; se R rifiuta, *accetta*.

Si noti che S deve effettivamente essere in grado di calcolare una descrizione di M_1 da una descrizione di M e w. È in grado di farlo perché ha solo bisogno di aggiungere stati extra a M che eseguono il test x = w.
Se R fosse un decisore per E_{TM}, S sarebbe un decisore per A_{TM}. Un decisore per A_{TM} non può esistere, quindi sappiamo che E_{TM} deve essere indecidibile.
Procediamo con l'esempio dello *stabilire se un linguaggio è regolare*:

$$REGULAR_{TM} = \{\langle M \rangle \mid M \text{ è una TM tale che } L(M) \text{ è regolare}\}$$

- La dimostrazione è per contraddizione e riduzione di A_{TM}
- Chiamiamo R la TM che decide $REGULAR_{TM}$
- Useremo R per costruire la TM S che decide A_{TM}
- Capire come possiamo usare R per implementare S è meno ovvio di prima

Come visto prima, la prova è per riduzione di A_{TM}. Si assuma che $REGULAR_{TM}$ è decidibile da una TM R e si usi l'assunzione per costruire una TM S che decide A_{TM}. L'idea è di usare S sull'input <M, w> e modificare M affinché la TM risultante riconosca un linguaggio regolare se e solo se M accetta w. Chiamiamo la macchina modificata M_2. Disegniamo M_2 per riconoscere il linguaggio non regolare $\{0^n1^n \mid n \geq 0\}$ se M non accetta "w" e per riconoscere il linguaggio regolare Σ* se M accetta "w". Dobbiamo specificare come S possa costruire una tale M_2 partendo da M e "w". Qui, M_2 funziona accettando automaticamente tutte le stringhe in $\{0^n1^n \mid n \geq 0\}$ e se M accetta "w", accetta tutte le altre stringhe.
Si noti che M_2 non è costruita per essere eseguita su qualche input, ma solamente per dare una descrizione per il decisore di $REGULAR_{TM}$ che abbiamo assunto esistere. Una volta che il decisore ritorna il suo risultato, possiamo usarlo per ottenere il risultato se M accetti "w". Perciò, possiamo decidere A_{TM}, che è contraddizione.

La dimostrazione si ha con R una TM che decide $REGULAR_{TM}$ e costruisca la TM S che decide A_{TM}. Allora S funziona nel seguente modo:
S = Sull'input <M, w>, dove M è una TM e w è una stringa:

1) Costruiamo la seguente TM M_2
   M_2 = "Sull'input x:
   1) Se x ha forma 0^n1^n, *accetta*
   2) Se x non ha questa forma, eseguire M sull'input w e *accetta* se M accetta w
2) Si esegue R sull'input <M_2>
3) Se R accetta, *accetta*; se R rifiuta, *rifiuta*

Allo stesso modo, i problemi di verificare se il linguaggio di una macchina di Turing è un linguaggio context-free, un linguaggio decidibile o anche un linguaggio finito possono essere dimostrati come indecidibili sulla base di dimostrazioni similari. Infatti, un risultato generale, chiamato *teorema di Rice*, afferma che determinare *qualsiasi proprietà* dei linguaggi riconosciuti dalle macchine di Turing è *indecidibile*.

$$EQ_{TM} = \{\langle M_1, M_2 \mid M_1, M_2 \text{ TM tali che } L(M_1) = L(M_2)\}$$

L'idea è di usare E_{TM} come Test del Vuoto e usare R che decida il linguaggio e la dimostrazione si struttura in questo modo.

D = "Su input <M>, dove M è una TM":

1) Esegue D su <M,M_1> e M_1 rifiuta tutte le stringhe
2) Similmente può scegliere di eseguire <M, M_2> su tutte le stringhe
3) Se R arriva ad uno stato di accettazione, allora *accetta*, altrimenti *rifiuta*

Dato che rifiutiamo tutte le stringhe che non fanno parte del linguaggio, allora correttamente la macchina si ferma accettando tutti gli input vuoti. Abbiamo dimostrato che esiste un decisore; tuttavia E_{TM} è indecidibile e allora $E_{TM} \leq_m EQ_{TM}$, ma E_{TM} è indecidibile e di conseguenze EQ_{TM} è indecidibile.

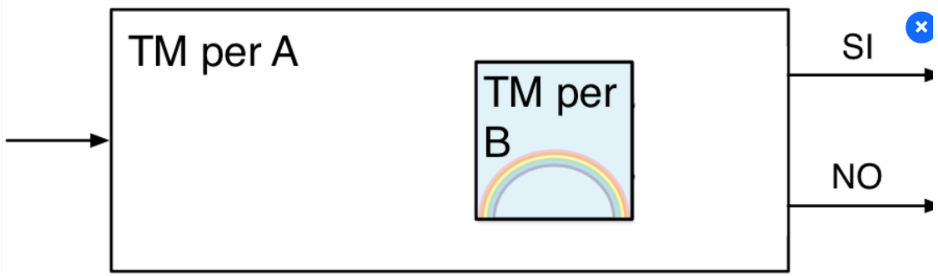
Domande Wooclap

Dato n, questa funzione termina?

```
void collatz(int n)
{
while(n > 1)
{
if(n % 2 == 1)
n = 3*n + 1;
else
n = n / 2;
}
return;
}
```

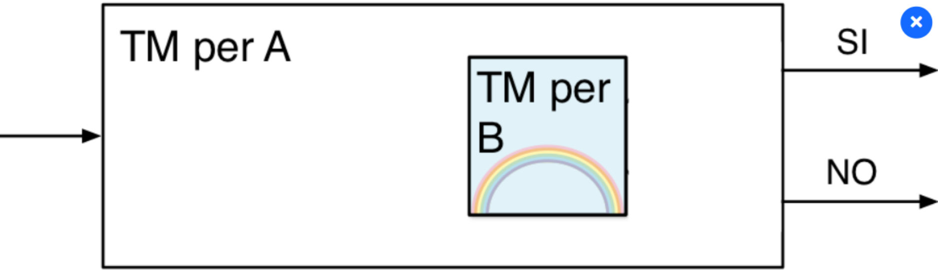
Risposta: Deriva da una congettura del matematico Collatz.
Maggiori al link: https://it.wikipedia.org/wiki/Congettura_di_Collatz
Risposta giusta quindi: *Non so*
Di fatto è un *halting problem* e si dimostra che in determinate condizioni, il calcolo di questa particolare successione può effettivamente fermarsi. D'altro canto, essa è un ciclo infinito descritto da una funzione e da un albero caratteristico.

Supponiamo che A sia riducibile a B: questo vuol dire che possiamo usare una soluzione per B per risolvere il problema A. Se B è *DECIDIBILE*, allora:



Risposta: A è decidibile

Supponiamo che A sia riducibile a B: questo vuol dire che possiamo usare una soluzione per B per risolvere il problema A. Se B è *INDECIDIBILE*, allora:



Risposta: A è indecidibile

Supponiamo che A sia riducibile a B: questo vuol dire che possiamo usare una soluzione per B per risolvere il problema A. Se A è *DECIDIBILE*, allora:

Risposta: Non possiamo dire nulla su B

Supponiamo che A sia riducibile a B: questo vuol dire che possiamo usare una soluzione per B per risolvere il problema A. Se A è *INDECIDIBILE*, allora:

Risposta: B è indecidibile

Sappiamo che A_{TM} è indecidibile. Possiamo usare questo fatto per dimostrare che anche $HALT_{TM}$ è indecidibile?

Risposta:

Si, costruendo una TM che risolve A_{TM} usando una TM per $HALT_{TM}$ come subroutine

Data una TM M e una parola w, costruiamo la seguente TM M_1:
$M_1 =$ "Su input x:
1. Se $x \neq w$, RIFIUTA
2. Se $x = w$, esegui M su w e ACCETTA se M accetta. Altrimenti RIFIUTA."
Quale delle seguenti affermazioni è vera?

Risposta:
$L(M_1) = \{w\}$ se M accetta w
$L(M_1) = \emptyset$ se M non accetta w

Data una TM M e una parola w, costruiamo la seguente TM M_2:
$M_2 =$ "Su input x:
1. Se x ha la forma 0^n1^n, ACCETTA
2. Se x non ha tale forma, esegui M su w e ACCETTA se M accetta w. Altrimenti RIFIUTA."
Quali delle seguenti affermazioni sono vere? (puoi scegliere più di una risposta)

Risposte:

$L(M_2) = \{0,1\}^*$ se M accetta W
$L(M_2) = 0^n1^n$ se M non accetta W

Completa la descrizione della TM S che decide E_{TM} usando la TM R che decide EQ_{TM}.
S deve accettare $\langle M \rangle$ se $L(M) = \emptyset$, e rifiutare quando $L(M) \neq \emptyset$

$S =$ "Su input $\langle M \rangle$, dove M è una TM:
1. ......

Riducibilità

Per cercare di risolvere e quantificare un problema, trasformando istanze del problema A in istanze del problema B mediante una *funzione calcolabile*, che ne chiarisce e formalizza la riducibilità.

$f : \Sigma^* \mapsto \Sigma^*$ è una funzione calcolabile se esiste una TM M che su input w, termina la computazione avendo solo $f(w)$ sul nastro

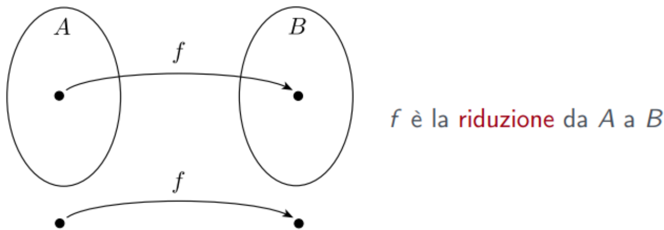
Tramite queste funzioni, operazioni come quelle aritmetiche eseguite sugli interi o anche le trasformazioni delle macchine di Turing possono essere funzioni calcolabili.

Viene così definita la *riducibilità mediante funzione*:

Definition

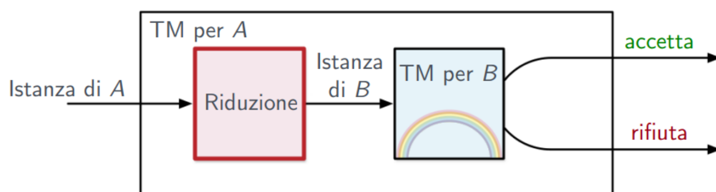
Un linguaggio A è riducibile mediante funzione al linguaggio B ($A \leq_m B$), se esiste una funzione calcolabile $f : \Sigma^* \mapsto \Sigma^*$ tale che

$$\text{per ogni } w : w \in A \text{ se e solo se } f(w) \in B$$



f è la riduzione da A a B

Se esiste una riduzione da A a B, possiamo risolvere A usando una soluzione per B:



Enunciamo due teoremi:

1) *Se $A \leq_m B$ e B è decidibile, allora A è decidibile.*

   Poniamo M decisore per B ed f la funzione di riduzione. Descriviamo un decisore per A nel modo seguente:

   M_A = su input w

   1. calcola f(w)
   2. esegue M_B su f(w) e ritorna lo stesso risultato di M_B

   Chiaramente se w ∈ A, f(w) ∈ B perché f è una riduzione da A B. Perciò N lavora come voluto.

Similmente:

2) *Se $A \leq_m B$ e A è indecidibile, allora B è indecidibile.*

Descriviamo il problema della fermata (*halting problem*), dimostrando che è indecidibile :

$$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ è una TM che si ferma su input } w\}$$

La funzione deve trasformare una macchina ed una stringa in una codifica di una macchina ed una stringa.

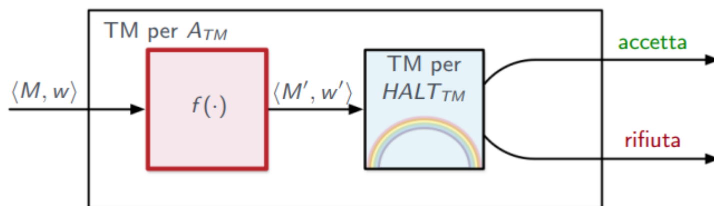
Di conseguenza, in fin dei conti, si ha una stringa generica:
<M, w> → <M', w'>
Per poter essere una funzione di riduzione:

- si deve avere corrispondenza tra input ed output
  (diamo corrispondenza al problema della fermata → $A_{TM} \leq_m HALT_{TM}$
  Formalmente: <M, w> ∈ A_{TM} se e solo se <M', w'> ∈ $HALT_{TM}$
- f è calcolabile

Passiamo quindi alla descrizione ad alto livello di una riduzione f e la macchina TM F.
F : Su input <M, w>

1) Costruisci la TM seguente M':
   M' : Su input x:
   a. Simula M su input x
   b. Se M accetta, *accetta*
   c. Se M rifiuta, va in loop
2) Ritorna <M', w>

Se F ha nel suo input qualcosa che non appartiene alle sue stringhe, dà in output una stringa che non sta in $HALT_{TM}$; pertanto quando si descrive una Turing machine che calcola una riduzione da A verso B, input non corretti sono stringhe mappate al di fuori di B.



M accetta w se e solo se M' si ferma su w'

Discutiamo similmente in merito al problema del vuoto:

$$E_{TM} = \{\langle M \rangle \mid M \text{ è una TM tale che } L(M) = \emptyset\}$$

In output ci sarà una TM M' in quanto il problema del vuoto prende <M, w> → <M'>

1) M accetta w sse L(M') = Ø
2) P è calcolabile

F : Su input <M, w> dove M è una TM e w è una stringa:

1) Costruisco la TM M' su input x:

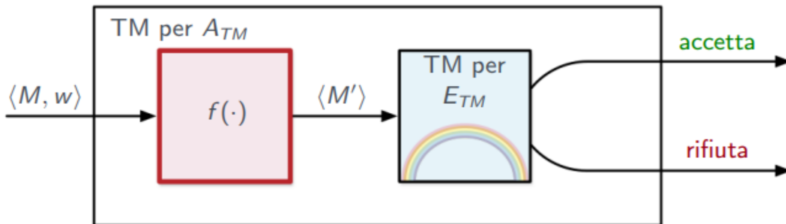
- se x ≠ w, *rifiuta*
- se x = w, simula M su w
- se M accetta, *rifiuta* altrimenti *accetta*

Output <M'>
Quindi la macchina non accetta nulla in queste condizioni, nel senso che il problema del vuoto attesta che sia la stringa stessa sia altre vengano tutte rifiutate. Tuttavia:
G: su input <M, w> M è TM e w è stringa:

1. Costruisci la TM:
   M' su input x:
   1. se x ≠ w, *rifiuta*
   2. se x = w, simula M su w
   3. se M accetta, *accetta* altrimenti *rifiuta*

Rappresenta l'opposto di ciò che volevamo dimostrare; questo perché:
<M, w> ∈ A_{TM} sse L(M') ≠ Ø
<M, w> ∉ A_{TM} sse L(M') = Ø
<M, w> ∈ $\underline{A_{TM}}$ sse L(M') = Ø

Il complementare si può quindi ridurre, mediante funzione, al problema del vuoto.
Dato che anche il complementare del problema dell'accettazione è indecidibile, allora anche il problema è indecidibile.



M accetta w se e solo se $L(M') = \emptyset$

Enunciamo altri due teoremi:
- Se $A \leq_m B$ e B è Turing-riconoscibile, allora A è Turing-riconoscibile
- Se $A \leq_m B$ ed A non è Turing-riconoscibile, allora B non è Turing-riconoscibile

Esaminiamo ora il *problema dell'equivalenza*:

$$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1, M_2 \text{ TM tali che } L(M_1) = L(M_2)\}$$

In questo caso potremmo dimostrare che EQ_{TM} non sia né Turing-riconoscibile né co-Turing-riconoscibile.
Vogliamo trovare un problema che si può ridurre tramite funzione al problema dell'equivalenza.
In particolare, si vuole un problema non Turing-riconoscibile, che sarebbe in questo caso il complementare dell'accettazione.
Quindi: $\overline{A_{TM}} \leq_m EQ_{TM}$
E in particolare: $A_{TM} \leq_m \overline{EQ_{TM}}$

Andremo ad applicare quindi: $A_{TM} \leq_m \overline{EQ_{TM}}$
Prendiamo quindi una funzione H che rispetta come proprietà:
$\langle M, w \rangle \in A_{TM}$ sse $\langle M_1, M_2 \rangle \in \overline{EQ_{TM}}$
La descrizione segue, partendo da una funzione di riduzione *f*:

H: su input $\langle M, w \rangle$ con M che è una TM e w che è una stringa:
1) si costruisce M_1 che su input x:
   1. *rifiuta*
2) si costruisce M_2 che su input x:
   1. Simula M su w
   2. Se M accetta, *accetta*
3) ritorna $\langle M_1, M_2 \rangle$

Qui, M_1 non accetta nulla. Se M accetta w, M_2 accetta tutto e le due macchine non sono equivalenti. Al contrario, se M non accetta w, M_2 non accetta nulla e sono equivalenti. Pertanto, f riduce A_{TM} ad $\overline{EQ_{TM}}$ come desiderato.
Di conseguenza vale che: $\langle M, w \rangle \in A_{TM}$ $L(M_1) = \emptyset$ $L(M_2) = \Sigma^*$ $\langle M_1, M_2 \rangle \in \overline{EQ_{TM}}$

Per dimostrare che $\overline{EQ_{TM}}$ non è Turing-riconoscibile, diamo una riduzione da A_{TM} al complemento di $\overline{EQ_{TM}}$.
Perciò mostriamo che:
$\langle M, w \rangle \notin A_{TM}$ $L(M_1) = \emptyset$ $L(M_2) = \emptyset$ $\langle M_1, M_2 \rangle \notin \overline{EQ_{TM}}$
e più concretamente:
$A_{TM} \leq_m EQ_{TM}$. La successiva TM H computa la funzione di riduzione *h*.

H: su input <M, w>, dove M è una TM e w è una stringa:

1) si costruisce M_1 che su input *x*:
   1. *accetta*
2) si costruisce M_2 che su input x:
   1. Simula M su w
   2. Ritorna lo stesso risultato di M'
3) ritorna <M_1, M_2>

L'unica differenza tra *f* e *g* è nella macchina M_1. In *f*, la macchina M_1 rifiuta sempre, mentre in *h* accetta sempre. In entrambe le funzioni, M accetta w se M_2 accetta sempre. In *h*, M accetta w se e solo se M_1 ed M_2 sono equivalenti. Pertanto, ecco perché *h* è riduzione di A_{TM} in EQ_{TM}.

<u>Risposte Wooclap</u>

Supponiamo che A sia riducibile mediante funzione a B. Questo vuol dire che esiste una funzione calcolabile che trasforma istanze di A in istanze di B. Quali delle seguenti affermazioni sono vere?

Risposte:

Se B è decidibile, allora A è decidibile

Se A è indecidibile, allora B è indecidibile

Per dimostrare che $A_{TM} \leq_m HALT_{TM}$ dobbiamo trovare una funzione calcolabile f che rispetta certe proprietà. Qual è l'INPUT della funzione di riduzione?

Risposta: Una TM e una stringa <M, w>

Sarebbe l'OUTPUT nella domanda:

Per dimostrare che $A_{TM} \leq_m HALT_{TM}$ dobbiamo trovare una funzione calcolabile f che rispetta certe proprietà. Qual è l'INPUT della funzione di riduzione?

Risposta: Una TM e una stringa <M, w>

Per dimostrare che $A_{TM} \leq_m E_{TM}$ dobbiamo trovare una funzione calcolabile f che rispetta certe proprietà. Qual è l'OUTPUT della funzione di riduzione?

Risposta: Una TM e una stringa <M, w>

Per dimostrare che $A_{TM} \leq_m E_{TM}$ dobbiamo trovare una funzione calcolabile f che rispetta certe proprietà. Qual è l'INPUT della funzione di riduzione?

Risposta: Una TM <M'>

Per dimostrare che $A_{TM} \leq_m E_{TM}$ dobbiamo trovare una funzione calcolabile f che rispetta certe proprietà. Qual è la relazione tra input e output della funzione di riduzione?

Risposta: M accetta w se e solo se L(M') = ∅

Supponiamo che A sia riducibile mediante funzione a B. Questo vuol dire che esiste una funzione calcolabile che trasforma istanze di A in istanze di B. Quali delle seguenti affermazioni sono vere?

Risposte:

Se A non è Turing-riconoscibile, allora B non è Turing-riconoscibile

Se B è Turing-riconoscibile, allora A è Turing-riconoscibile

Quale riduzione possiamo usare per dimostrare che EQ_{TM} **non è Turing-riconoscibile?**

Risposta:

$$\overline{A_{TM}} \leq_m EQ_{TM}$$

Quale riduzione possiamo usare per dimostrare che EQ_{TM} **non è coTuring-riconoscibile?**

Risposta:

$A_{TM} \leq_m EQ_{TM}$

Complessità di tempo/Classe P

Consideriamo come linguaggio $A = \{0^k 1^k \mid k \geq 0\}$

Vogliamo capire che tipo di linguaggio sia, se sia decidibile e quanto tempo serve per una TM a nastro singolo per decidere il linguaggio. Definiamo quindi:

- Sia M una TM deterministica che si ferma su tutti gli input.
- Il tempo di esecuzione (o complessità di tempo) di M è la funzione $f : \mathbb{N} \mapsto \mathbb{N}$ tale che $f(n)$ è il numero massimo di passi che M utilizza su un input di lunghezza n.
- Se $f(n)$ è il tempo di esecuzione di M, diciamo che M è una TM di tempo $f(n)$.
- Useremo n per rappresentare la lunghezza dell'input.
- Ci interesseremo dell'analisi del caso pessimo.

Usiamo la notazione *O-grande* che valuta il tempo di esecuzione su input grandi e considera solo il termine di ordine maggiore e ignora i coefficienti.

Date due funzioni f, g, diciamo che $f(n) = O(g(n))$ se esistono interi positivi c, n_0 tali che per ogni $n \geq n_0$:

$$f(n) \leq c\, g(n).$$

$g(n)$ è un limite superiore asintotico per $f(n)$

Analizziamo questa TM per $A = \{0^k 1^k \mid k \geq 0\}$

M_1 = "Su input w:

1. Scorre il nastro e rifiuta se trova uno 0 a destra di un 1
2. Ripete finché il nastro contiene almeno uno 0 ed un 1:
3. Scorre il nastro cancellando uno 0 ed un 1
4. Se rimane almeno uno 0 dopo che ogni 1 è stato cancellato, o se rimane almeno un 1 dopo che ogni 0 è stato cancellato, rifiuta. Altrimenti, se non rimangono né 0 né 1 sul nastro, accetta"

Per analizzare M_1 consideriamo le quattro fasi separatamente.

- Nella prima fase la TM scansione se l'input è nella forma 0*1*, mettendoci *n* passi (n = lunghezza dell'input). Similmente, per riposizionare la testina occorrono altri *n* passi. Quindi siamo con un numero di passi = 2n
- Nelle fasi due e tre la TM scansiona il nastro attraversandolo e capendo se si tratta di uno 0 o di un q, mettendoci al più n/2 passi (perché il nastro scorre linearmente per 2, dunque ci mette la metà dei passi). Pertanto $(n/2)O(n) = O(n^2)$
- Nella fase quattro, la macchina impiega un singolo scan per capire se decide o meno il linguaggio. Dunque, il tempo che ci impiega è lineare e sarà O(n). In totale, comunque, si decide tutto in tempo $O(n) * O(n/2) = O(n^2)$

Sia $t : \mathbb{N} \mapsto \mathbb{N}$ una funzione.

La classe di complessità di tempo $\text{TIME}(t(n))$ è l'insieme di tutti i linguaggi che sono decisi da una TM in tempo $O(t(n))$.

- Questo è diverso dalle classi di linguaggi discusse in precedenza, che si concentravano sulla computabilità.
- Questa classificazione si concentra sul tempo necessario per decidere il linguaggio.

Dall'analisi fatta, sappiamo che $A = \{0^k 1^k \mid k \geq 0\}$ ed appartiene a $\text{TIME}(n^2)$ perché M_1 decide A in tempo $O(n^2)$. Possiamo fare di meglio?
Si può costruire una TM multinastro simile al funzionamento della pila.
L'idea è di cancellare metà degli 0 e metà degli 1 ad ogni scansione.
In particolare, abbiamo:
M_2 = "Su input *w*, dove *w* è una stringa:"

1. Scorre il nastro e rifiuta se trova uno 0 a destra di un 1
2. Ripete finché il nastro contiene almeno uno 0 ed un 1:
   3. Scorre il nastro e controlla se il numero totale di 0 e 1 rimasti è pari o dispari. Se è dispari, rifiuta.
   4. Scorre il nastro, cancellando prima ogni secondo 0 a partire dal primo 0, poi cancellando ogni secondo 1 a partire dal primo 1.
5. Se nessuno 0 e nessun 1 rimangono sul nastro, accetta. Altrimenti rifiuta."

Prima di analizzare M_2, verifichiamo che decida effettivamente A. In ogni scansione eseguita nella fase 4, il numero totale di 0 rimanenti viene dimezzato e qualsiasi resto viene scartato. Quindi, se abbiamo iniziato con 13 zeri, dopo che la fase 4 è stata eseguita una sola volta, rimangono solo 6 zeri. Dopo le successive esecuzioni di questa fase, ne rimangono 3, poi 1 e poi 0. Questa fase ha lo stesso effetto sul numero di 1.

Ora esaminiamo la parità pari/dispari del numero di 0 e il numero di 1 ad ogni esecuzione della fase 3. Considera di nuovo di iniziare con 13 zeri e 13 uni. La prima esecuzione della fase 3 trova un numero dispari di 0 (perché 13 è un numero dispari) e un numero dispari di 1. Nelle esecuzioni successive, si verifica un numero pari (6), quindi un numero dispari (3) e un numero dispari (1). Non eseguiamo questa fase su 0 zeri o 0 uni a causa della condizione sul ciclo di ripetizione specificata nella fase 2. Per la sequenza di parità trovate (dispari, pari, dispari, dispari), se sostituiamo i pari con 0 e le quote con 1 e poi invertiamo la sequenza, otteniamo 1101, la rappresentazione binaria di 13, o il numero di 0 e 1 all'inizio. La sequenza di parità dà sempre il contrario della rappresentazione binaria.

Per analizzare il tempo di esecuzione, ci mette anche qui *n* passi a scansionare l'input e tutte le fasi, scorrendo linearmente il nastro, impiegato un tempo O(n). La fase 4, tuttavia, attraversa almeno metà degli 0 ed 1 ogni volta che viene eseguito, in totale per un tempo $1 + \log_2(n)$, immaginando che ne esista almeno 1 (da cui l'uno) ed eseguendo una scansione a metà per ogni iterazione lineare (da cui il log).

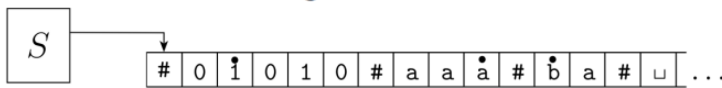
Perciò il tempo totale di esecuzione sarà $O(n) + (1 + log2(n)) = O(n) + O(n\, log(n) = O\,(n\, log(n))$

Possiamo quindi trovare una TM che decide A in O(n)?
Non esiste una TM a nastro singolo in grado di decidere A in tempo O(n); possiamo farlo se la TM ha un secondo nastro. La complessità di tempo dipende dal modello di calcolo.
La tesi di Church-Turing implica che tutti i modelli di calcolo siano ragionevolmente equivalenti.

In merito alla differenza tra TM a singolo nastro e TM multinastro (determinando la simulazione della macchina singola con la macchina multinastro):

Sia $t(n)$ una funzione tale che $t(n) \leq n$. Ogni TM multinastro di tempo $t(n)$ ammette una TM equivalente a nastro singolo di tempo $O(t^2(n))$.

- Ricordiamo la dimostrazione di come convertire una TM da multinastro a nastro singolo.



- Dobbiamo determinare quanto tempo ci vuole per simulare ogni passo della macchina multinastro sulla TM a nastro singolo.

avendo le TM a nastro singolo che risolvono lo stesso problema in tempo quadratico.
Ci ricordiamo l'equivalenza tra le TM a nastro singolo e le TM multinastro (che va fatta dalle multinastro a quelle a nastro singolo aggiungendo un simbolo delimitatore e poi fisicamente operando sul nastro, il contrario invece si prende un delimitatore e si estende la computazione della TM a nastro singolo eseguendo *k* operazioni sui vari nastri multipli).

Possiamo quindi decidere A in un tempo lineare se la TM ha un secondo nastro.
S: su input w

O(n) *k+n*	1) scrive #w_1* $w_{2\ldots}$ w_n # |_|*#...... # |_|*#
t(n) volte:	2) Simula un passo di M
k*t(n)	a. Scorri il nastro per determinare il simbolo sotto le testine virtuali
k*t(n)	b. Aggiorna il nastro spostando le testine virtuali come dettato dalla funzione di transizione di M
O(1)	c. se M accetta, *accetta*, se M rifiuta, *rifiuta*. Altrimenti ripeto da 2.

O(n) *k+n* per t(n) volte per vedere dove si trovano le testine virtuali.
t(n) è il tempo di calcolo per la TM multinastro per decidere il linguaggio.

La seguente macchina M_3 decide A e opera in modo diverso, in quanto semplicemente copia gli 0 sul secondo nastro e cerca di farli corrispondere al numero di uni.
M_3: su input w
- 1. Esegue la scansione sul nastro 1 e rifiuta se viene trovato uno 0 a destra di un 1.
- 2. Scansiona gli 0 sul nastro 1 fino al primo 1. Allo stesso tempo, copia gli 0 sul nastro 2.
- 3. Esegue la scansione degli 1 su nastro 1 fino alla fine dell'input. Per ogni 1 lettura sul nastro 1, cancellare uno 0 sul nastro 2. Se tutti gli 0 vengono cancellati prima che tutti gli 1 vengano letti, rifiuta.
- 4. Se tutti gli 0 sono stati cancellati, *accetta*. Se rimangono degli 0, *rifiuta*.

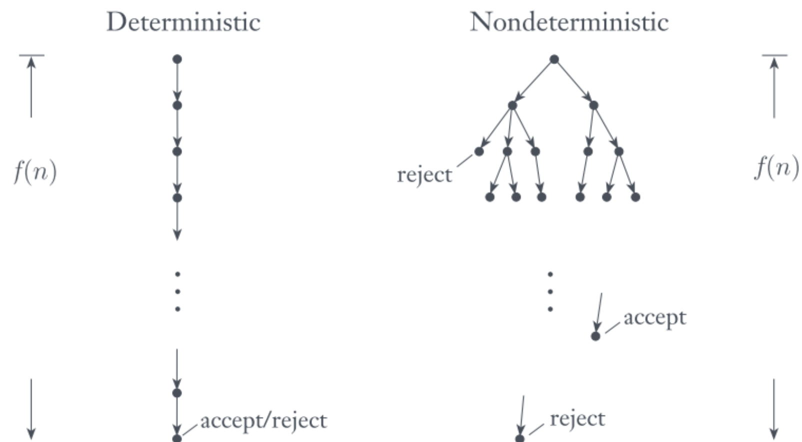
Nel caso di *TM non deterministiche*:

Sia N una TM non deterministica che è anche un decisore. Il tempo di esecuzione di N è la funzione $f : \mathbb{N} \mapsto \mathbb{N}$ tale che $f(n)$ è il massimo numero di passi che N usa per ognuno dei rami di computazione, su input di lunghezza n.

La definizione di tempo di esecuzione per le TM non deterministiche non è destinata corrispondere ad un qualche dispositivo di calcolo reale. È uno strumento teorico che utilizziamo per comprendere i problemi computazionali.

Pensiamo che una TM non deterministica sia in grado di capire quale sia il percorso da prendere e, randomicamente può avere un tempo di esecuzione lineare.

Prendiamo ora un confronto tra determinismo e non determinismo:



Seguendo il libro, dice che la definizione è per $t(n) \geq n$.

Sia $t(n)$ una funzione tale che ~~$t(n) \leq n$~~. Ogni TM non deterministica di tempo $t(n)$ ammette una TM equivalente a nastro singolo di tempo $2^{O(t(n))}$.

Dimostrazione:
Sia N una TM non deterministica che esegue in tempo lineare. Costruiamo una TM D deterministica che simula N cercando l'albero di calcolo non deterministico di N. Ora analizziamo quella simulazione. Su un input di lunghezza n, ogni ramo dell'albero di calcolo non deterministico di N ha una lunghezza di al massimo t(n). Ogni nodo dell'albero può avere al massimo b figli, dove b è il numero massimo di scelte legali date dalla funzione di transizione di N. Pertanto, il numero totale di foglie nell'albero è al massimo $b^{t(n)}$. La simulazione procede esplorando prima l'albero in modo breadth first. In altre parole, visita tutti i nodi a profondità *d* prima di passare a uno qualsiasi dei nodi a profondità *d + 1*. L'algoritmo inizia in modo inefficiente dalla radice e viaggia verso un nodo ogni volta che visita quel nodo. Il numero totale di nodi nell'albero è meno del doppio del numero massimo di foglie, quindi è limitato da $O(b^{t(n)})$.
Il tempo necessario per iniziare dalla radice e viaggiare fino a un nodo è O(t(n)).
Pertanto, il tempo di esecuzione di D è $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

Se abbiamo una TM non deterministica N in tempo T(n), si agisce per livelli spostandosi in profondità verso ciascuno, arrivando ai nodi di profondità massima.
Dato un albero binario si ha $2^n - 1 = 2^{t(n)} - 1$
E: $3^{t(n)} - 1 \approx 2^{O(t(n))}$

Quindi:
- Differenza di tempo polinomiale tra TM a nastro singolo e multi-nastro
- Differenza di tempo esponenziale tra TM deterministiche e non deterministiche.

Una differenza *polinomiale* è considerata piccola e tutti i modelli di calcolo ragionevoli (molto somiglianti ai computer reali) sono *polinomialmente equivalenti*.
Una differenza *esponenziale* è considerata grande, simile alla complessità brute force/a forza bruta.

Ciò che possiamo dire è che:

P è la classe di linguaggi che sono decidibili in tempo polinomiale da una TM deterministica a singolo nastro:

$$P = \bigcup_k \mathrm{TIME}(n^k)$$

P è invariante per i modelli di calcolo polinomialmente equivalenti ad una TM deterministica e corrisponde approssimativamente ai problemi che sono realisticamente risolvibili da un computer.
Per dimostrare che un problema/algoritmo è in P:

- Descrivi l'algoritmo per fasi numerate
- Dai un limite superiore polinomiale al numero di fasi che l'algoritmo esegue per un input di lunghezza n
- Assicurati che ogni fase possa essere completata in tempo polinomiale su un modello di calcolo deterministico ragionevole
- L'input deve essere codificato in modo ragionevole

Affrontiamo questi due problemi:

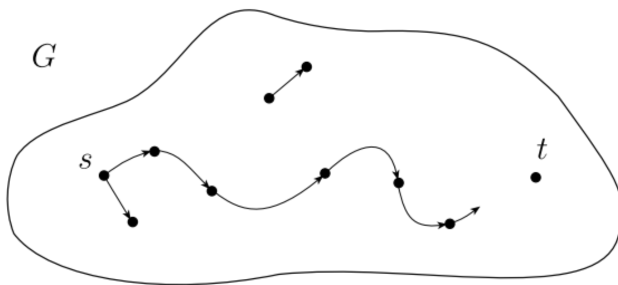
Raggiungibilità in un grafo

$PATH = \{\langle G, s, t\rangle \mid G \text{ grafo che contiene un cammino da } s \text{ a } t\}$

Numeri relativamente primi

$RELPRIME = \{\langle x, y\rangle \mid 1 \text{ è il massimo comun divisore di } x \text{ e } y\}$

Per il problema PATH:



Prima di descrivere questo algoritmo, osserviamo che un algoritmo brute force per questo problema non è abbastanza veloce. Un algoritmo di forza bruta per PATH procede esaminando tutti i potenziali percorsi in G e determinando se uno di essi sia un percorso diretto da s a t. Un potenziale percorso è una sequenza di nodi in G di lunghezza al massimo pari a m, dove m è il numero di nodi in G.
(Se esiste un percorso diretto da s a t, ne esiste uno di lunghezza al massimo m, perché la ripetizione di un nodo non è mai necessaria). Ma il numero di tali percorsi potenziali è circa m^m, che è esponenziale rispetto al numero di nodi in G. Pertanto, questo algoritmo di forza bruta richiede un tempo esponenziale.
Per ottenere un algoritmo in tempo polinomiale per PATH, dobbiamo fare qualcosa che eviti la brute force. Un modo è quello di utilizzare un metodo di ricerca su grafo come la ricerca breadthfirst. In questo caso, contrassegniamo successivamente tutti i nodi di G che sono raggiungibili da s con percorsi diretti di lunghezza 1, poi 2, poi 3, attraverso m. Il tempo di esecuzione con strategia polinomiale è semplice.

Dimostrazione: Su M che su input <G, s, t>, dove G è grado diretto di nodi *s* e *t*:

O(n) + $O(n^3)$ + O(n) = $O(n^3)$

P = su input <G, s, t>:
1) marca il nodo S
2) ripeti finché riesci a marcare nuovi nodi
3) marca ogni vertice di G. Se un vertice (a, b) viene trovato spostandosi da un nodo marcato *a* ad un nodo non marcato *b*, marca il nodo *b*
4) se t è marcato, *accetta*, altrimenti *rifiuta*

Analizziamo ora questo algoritmo per dimostrare che funziona in tempo polinomiale.
Ovviamente, le fasi 1 e 4 vengono eseguite una sola volta. La fase 3 viene eseguita al massimo m volte perché ogni volta, tranne l'ultima, contrassegna un nodo aggiuntivo in G. Pertanto, il numero totale di stadi utilizzati è al massimo 1 + 1 + m, quindi un polinomio nella dimensione di G.
Gli stadi 1 e 4 di M sono facilmente implementabili in tempo polinomiale su qualsiasi modello deterministico ragionevole. La fase 3 comporta una scansione dell'input e un test per verificare se alcuni nodi sono se alcuni nodi sono marcati, anch'esso facilmente implementabile in tempo polinomiale.
Quindi M è un algoritmo in tempo polinomiale per PATH.
Ho un insieme di operazioni eseguite per $O(n) + O(n^3) + O(n) = O(n^3)$

Per il problema RELPRIME:

Idea:
Un algoritmo che risolve questo problema cerca tutti i possibili divisori di entrambi i numeri e accetta se nessuno di essi è maggiore di 1. Tuttavia, la grandezza di un numero rappresentato in binario, o in qualsiasi altra notazione in base k per k ≥ 2, è esponenziale rispetto alla lunghezza della sua rappresentazione.
Pertanto, questo algoritmo brute force ricerca un numero esponenziale di potenziali divisori e ha un tempo di esecuzione esponenziale.
Risolviamo invece questo problema con un'antica procedura numerica, chiamata *algoritmo di Euclide*, per calcolare *il divisore comune più grande* dei numeri naturali x e y, scritto $gcd\ (x, y)$, è il più grande numero intero che divide equamente. Il massimo comun divisore dei numeri naturali x e y, scritto $gcd(x, y)$, è il più grande numero intero che divide uniformemente x e y. Per esempio, $gcd(18, 24) = 6$.
Ovviamente x e y sono relativamente primi se $gcd\ (x, y) = 1$.
Nella dimostrazione, E è l'algoritmo di Euclide.

Dimostrazione:

O (log(x+y))
O ($\log(2^n)$)
O (n)

E = Su input <x, y>:
1) ripeti finché x ≠ y e in particolare finché y = 0
2) Assegna a *x* ← *x mod y*
3) Scambia x e y
4) Manda in output *x*

L'algoritmo R risolve poi RELPRIME usando E come subroutine.
R = Su input <x, y>, dove x e y sono numeri naturali:
1) Esegue E su <x, y>
2) Se il risultato è 1, *accetta*, altrimenti *rifiuta*

Per analizzare la complessità temporale di E, dimostriamo innanzitutto che ogni esecuzione della fase 2 (tranne forse la prima) riduce il valore di x di almeno la metà.
Dopo l'esecuzione della fase 2, *x è minore di y (x < y)* a causa della natura della funzione *mod*.
Dopo l'esecuzione della fase 2, *x è maggiore di y (x > y)* perché i due valori sono stati scambiati.
Pertanto, quando la fase 2 viene successivamente eseguita, x > y.
Se x/2 ≥ y, allora x mod y < y ≤ x/2 e x diminuisce di almeno la metà.
Se x/2 < y, allora x mod y = x - y < x/2 e x si riduce di almeno la metà.

I valori di x e y vengono scambiati ogni volta che viene eseguita la fase 3, quindi ciascuno dei valori originali di x e y si riduce di almeno la metà ogni volta che viene eseguito il ciclo. Pertanto, il numero massimo di volte in cui vengono eseguiti gli stadi 2 e 3 è il minore tra $2\log_2(x)$ e $2\log_2(y)$.
Questi logaritmi sono proporzionali alle lunghezze delle rappresentazioni, per cui il numero di stadi eseguiti è O(n). Ciascuna fase di E impiega solo un tempo polinomiale, quindi il tempo di esecuzione totale è polinomiale. La complessità è logaritmica rispetto alla somma di x + y.

Un'altra nota:

Ogni linguaggio context-free è un elemento di P.

- Abbiamo dimostrato che ogni CFL è **decidibile**
  - l'algoritmo nella dimostrazione è **esponenziale**
- La soluzione polinomiale usa la **programmazione dinamica**
- La complessità è $O(n^3)$

Risposte Wooclap

Considera il linguaggio $A = \{0^n 1^n \mid n \geq 0\}$. Quanto tempo serve ad una Macchina di Turing per decidere il linguaggio?

The total time of M_1 on an input of length n is $\mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$

Indica quali delle seguenti affermazioni sono vere. Puoi scegliere più di una risposta.

La risposta corretta era

Per la terza risposta

With some algebra (and changing the constant in the $O(n)$), we can

$\log_2 3)^n = 2^{n \log_2 3}$

$3 = O(n)$. So $3^n = 2^{O(n)}$.

3^n grows faster than any exponential
$o(3^n)$ of course but it seems you mean
ess is that your statement applies to
u multiply the base by a constant, as
ltiply the number in the exponent by a

Rispondete alla domanda di Zio Paperone: quanti chicchi di riso ci sono sulla scacchiera alla fine dei raddoppi?

Risposta:
$2^{64} - 1$

Questa TM decide PATH in tempo polinomiale?

P = "su input $\langle G, s, t \rangle$, dove G è un grafo, s, t, sono vertici di G:
1. Considera tutti i cammini che iniziano da s di lunghezza crescente $1, 2, \ldots, n$, dove n è il numero di vertici del grafo.
2. Se uno dei cammini termina in t, ACCETTA
3. Se nessuno dei cammini termina in t, RIFIUTA."

Partendo da n
1 * (n – 1) * (n – 2) * (n – 3) ……… 2 * 1
È un algoritmo più che polinomiale

Classe NP

Si parte dal gioco del domino, disponendole in file tale che si possano usare tutte le tessere.
Vogliamo capire se questo problema può essere facile o difficile.
Un problema è *trattabile* se esiste un *algoritmo efficiente* in grado di risolverlo, in particolare in tempo polinomiale $O(n^k)$ con un certo algoritmo per qualche costante *k* e risolvibile tramite una TM.
L'obiettivo quindi è di:

- Trovare un algoritmo polinomiale per Domino[1]

Possiamo definire il problema come un linguaggio e tale che sia risolvibile attraverso una riduzione.

$$D_1 = \{\langle B \rangle \mid B \text{ è un insieme di tessere del domino, ed esiste un allineamento che usa tutte le tessere}\}$$

- Usiamo una riduzione mediante funzione per trovare l'algoritmo polinomiale
- Riduciamo D_1 ad un problema su grafi ...
- ... per il quale sappiamo che esiste un algoritmo polinomiale

Come già detto, definiamo un grafo non orientato G tale che sia formato da una coppia (V,E) descrivendo così le tessere:

- $V = \{v_1, v_2, \ldots, v_n\}$ è un insieme finito e non vuoto di vertici;
- $E \subseteq \{\{u, v\} \mid u, v \in V\}$ è un insieme di coppie non ordinate, ognuna delle quali corrisponde ad un arco del grafo.

Con i vertici colleghiamo i numeri descritti dalle tessere:

Grafo del domino

- Vertici: i numeri che si trovano sulle tessere
  - V = {⚀, ⚁, ⚂, ⚃}
- Archi: le tessere del domino
  - E = {⚂⚃, ⚀⚁, ⚃⚁, ⚃⚀, ⚂⚁}

Vogliamo quindi ridurre questo problema ad uno di tipo euleriano, problema noto in teoria dei grafi, tale da poterlo risolvere in tempo polinomiale trovandone un cammino.

- **Cammino Euleriano**: percorso in un grafo che attraversa tutti gli archi una sola volta

Il problema del Cammino Euleriano

$EULER = \{\langle G \rangle \mid G$ è un grafo che possiede un cammino Euleriano$\}$

- *EULER* è un problema classico di teoria dei grafi
- Esistono algoritmi polinomiali per risolverlo

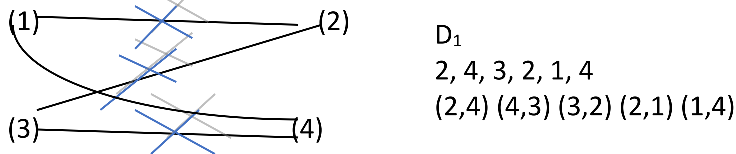
L'algoritmo che lo risolve è il cosiddetto *algoritmo di Fleury* così descritto:

1. Scegliere un vertice con grado dispari (un vertice qualsiasi se tutti pari)
2. Scegliere un arco tale che sua cancellazione non sconnetta il grafo
3. Passare al vertice nell'altra estremità dell'arco scelto
4. Cancellare l'arco dal grafo
5. Ripetere i tre passi precedenti finche non eliminate tutti gli archi

Complessità

Su un grafo con n archi, l'algoritmo di Fleury impiega tempo $O(n^2)$

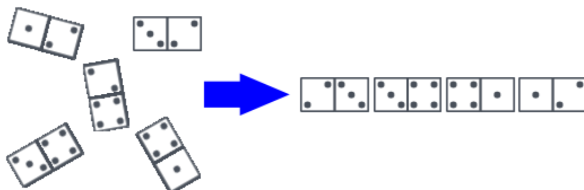
Ammettiamo di ragionare sul grafo precedente:



Quindi $D_1 \leq_M$ EULER tramite la riduzione dell'algoritmo di Fleury in tempo polinomiale.
Il tempo di trasformazione è lineare e quindi la somma permane in una complessità lineare e anche polinomiale.

Disponiamo per Domino[2], leggera variante con la seguente caratteristica e vogliamo sapere se può essere un problema facile o difficile da risolvere:

2. in modo che ogni numero compaia esattamente due volte (potete usare meno tessere di quelle che avete).



Scritto da Gabriel

Questa volta operiamo una riduzione ma in senso opposto, usando il grafo hamiltoniano, tale che ogni vertice compaia esattamente due volte:

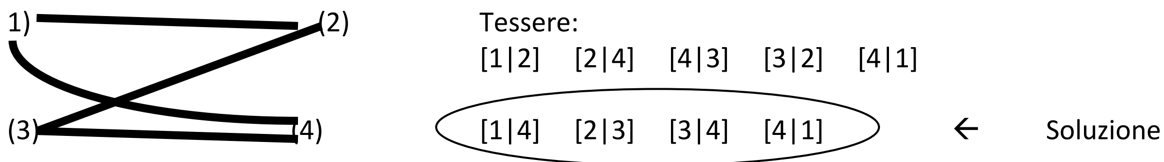
$D_2 = \{\langle B \rangle \mid B$ insieme di tessere del domino, ed esiste allineamento dove ogni numero compare due volte$\}$

- **Circuito Hamiltoniano**: ciclo nel grafo che attraversa tutti i vertici una sola volta

Il problema del Circuito Hamiltoniano

$HAMILTON = \{\langle G \rangle \mid G$ è un grafo con un circuito Hamiltoniano$\}$

Per dimostrare che HAMILTON $\leq_m$ D_2, possiamo prendere un qualsiasi grafo e ridurlo in una serie di tessere tali che, se in queste ogni numero compare due volte, allora esiste un circuito hamiltoniano.
Prendiamo il grafo precedente e per ogni arco costruisco una tessera:



Il problema del circuito Hamiltoniano è un problema classico di teoria dei grafi.
Un algoritmo polinomiale per risolverlo non è mai stato trovato; se qualcuno mi dà una possibile soluzione, è facile verificare se è corretta. Anche se esistesse un algoritmo polinomiale, essendo comunque trattando in tempo lineare, si avrebbe un algoritmo lineare per il circuito hamiltoniano.
Non sappiamo se esista questo algoritmo; tuttavia per questi problemi, *avendo una soluzione*, è facile *controllare se sia corretta* (tipo il caso tessere, vedere se sono allineate correttamente).

Come detto:
- i problemi per i quali esiste un algoritmo polinomiale vengono considerati *trattabili*
- quelli che richiedono un algoritmo più che polinomiale sono detti *intrattabili*

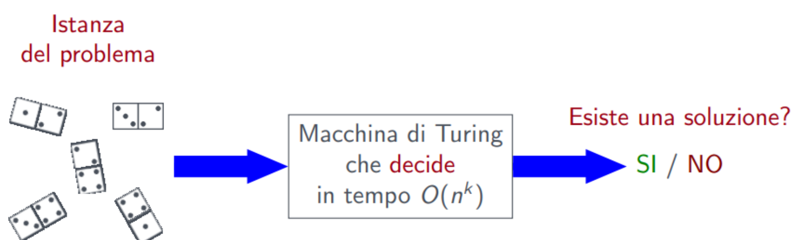
Sappiamo che ci sono problemi che non possono essere risolti da nessun algoritmo:
- "Halting Problem" di Turing

Sappiamo che ci sono problemi che possono essere risolti in tempo esponenziale:
- il gioco della Torre di Hanoi

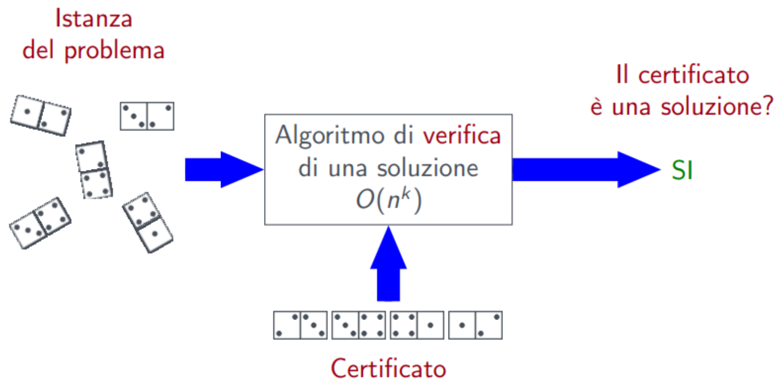
Stabilire con precisione qual'è il confine tra problemi trattabili ed intrattabili è piuttosto difficile

Classi di problemi:
P – Esiste decisore in tempo polinomiale → Facili da risolvere e da verificare
(Esempi: Domino[1], Euler, Relprime…)
NP – Esiste decisore in tempo polinomiale → Non facili da risolvere, potenzialmente mai risolvibili in poco tempo, ma facili da verificare
(Esempi: Hamilton, Sudoku, Protein folding, Crittografia…)

Posso quindi trovare nel caso di Domino[1]:



In merito invece a Domino[2], aggiungo un input al problema (*certificato*), semplificando il problema e sperando di ottenere un algoritmo polinomiale:



Possiamo quindi trovare un *verificatore*, quindi un algoritmo che:

Un verificatore per un linguaggio A è un algoritmo V tale che

$$A = \{w \mid V \text{ accetta } \langle w, c\rangle \text{ per qualche stringa } c\}$$

- il verificatore usa ulteriori informazioni per stabilire se w appartiene al linguaggio
- questa informazione è il certificato c

Quindi: un verificatore usa in input il problema stesso e un'informazione (*certificato*), tale da capire se *w* appartiene o meno ad un linguaggio.
In particolare:

- P è la classe dei linguaggi che possono essere decisi da una macchina di Turing deterministica che impiega tempo polinomiale.
- NP è la classe dei linguaggi che ammettono un verificatore che impiega tempo polinomiale.
  Definizione equivalente di NP: La classe dei linguaggi che possono essere decisi da una macchina di Turing non deterministica che impiega tempo polinomiale.

La macchina non deterministica prova, non deterministicamente, tutti i possibili certificati, indovinando la soluzione del problema e verificando se sia corretta. Non sappiamo se possa essere non polinomiale.

Vediamo un paio di esempi che sappiamo essere in NP:

Problema del circuito Hamiltoniano

$$HAMILTON = \{\langle G\rangle \mid G \text{ è un grafo con un circuito Hamiltoniano}\}$$

Numeri composti

$$COMPOSITES = \{\langle x\rangle \mid x = pq, \text{ per gli interi } p, q > 1\}$$

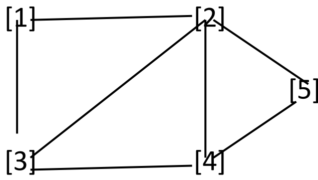
Considero HAMILTON = {<G> | G è un grafo con un circuito Hamiltoniano}
(tale da capire se <G> → SI/NO
Su input <G, c> con *G* che è un grafo e *c* è una sequenza di vertici:

1) Si verifica che tutti i vertici in *c* sono vertici del grafo
2) Controlla che ogni vertice di G compaia una sola volta
3) Controlla che *c* sia un cammino in G, C = $C_1, C_2, \dots, C_n$ $(C_i, C_{i+1}) \in E$
4) Controlla che sia un ciclo, cioè che $(C_n, C_1) \in E$

<u>Dal Sipser:</u>
N_1 = "Su input <G, s, t>, dove G è un grafo diretto con nodi s e t:
1. Scrive una lista di m numeri, $p_1, \dots, p_m$, dove m è il numero di nodi in G. di nodi in G. Ciascun numero della lista è scelto in modo non deterministicamente selezionato per essere compreso tra 1 e m.
2. Verifica la presenza di ripetizioni nell'elenco. Se ne vengono trovate, rifiuta.
3. Verifica se $s = p_1$ e $t = p_m$. Se entrambi falliscono, rifiuta.
4. Per ogni i compreso tra 1 e m - 1, verificare se (p_i, p_{i+1}) è un bordo di G. Se non lo è, rifiuta. Altrimenti, tutti i test sono stati superati, quindi accetta.

5) Se tutti i controlli sono positivi, *accetta*, altrimenti *rifiuta*

G= (scorrendo tutto il verificatore per il grafo)



Ora esaminiamo COMPOSITES = {<x> | x = pq, per gli interi p, q > 1}

1) verifica che p > 1
2) verifica che p < x
3) verifica che x sia divisibile per p
4) se è tutto OK, *accetta*, altrimenti *rifiuta*

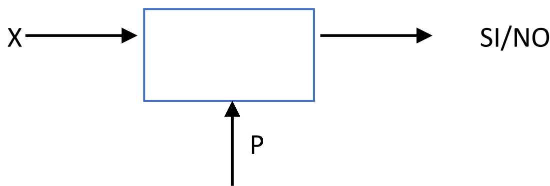
La seguente TM è un verificatore per *COMPOSITES*. Il certificato è uno dei due divisori di x

V = "su input $\langle x, p \rangle$, con x, p numeri interi:

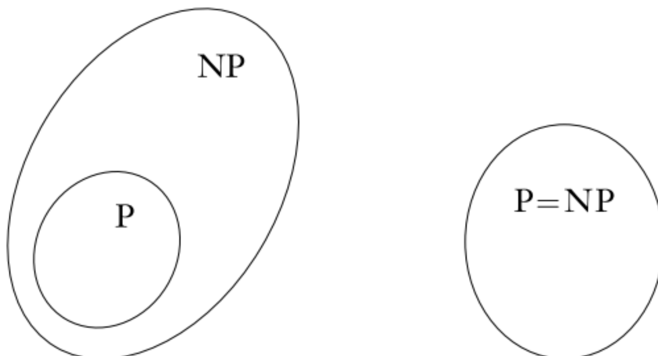
1. se $p \leq 1$ o $p = x$, rifiuta
2. se p è un divisore di x, accetta, altrimenti rifiuta."

Passo quindi dal decisore che dà in output SI oppure NO; in questo modo, il verificatore riuscirà a capire se trattasi di problema risolvibile o meno.

Quindi:



La relazione tra P ed NP è tuttora un grande problema non risolto; graficamente P sta in NP, ma questi ultimi non risolvono. Il potere della verificabilità polinomiale sembra essere molto maggiore di quello della decidibilità polinomiale. Ma, per quanto possa essere difficile da immaginare, P e NP potrebbero essere uguali. Non siamo in grado di dimostrare l'esistenza di una singola lingua in NP che non sia in P.
Capire se effettivamente P = NP è uno dei più grandi problemi irrisolti nell'informatica teorica e nella matematica contemporanea. Se queste classi fossero uguali, qualsiasi problema polinomialmente verificabile sarebbe polinomialmente decidibile. La maggior parte dei ricercatori ritiene che le due classi non siano uguali, visti gli sforzi senza successo nel trovare algoritmi utili e decisori.



Risposte Wooclap

Domino[1] è un problema che appartiene a P?

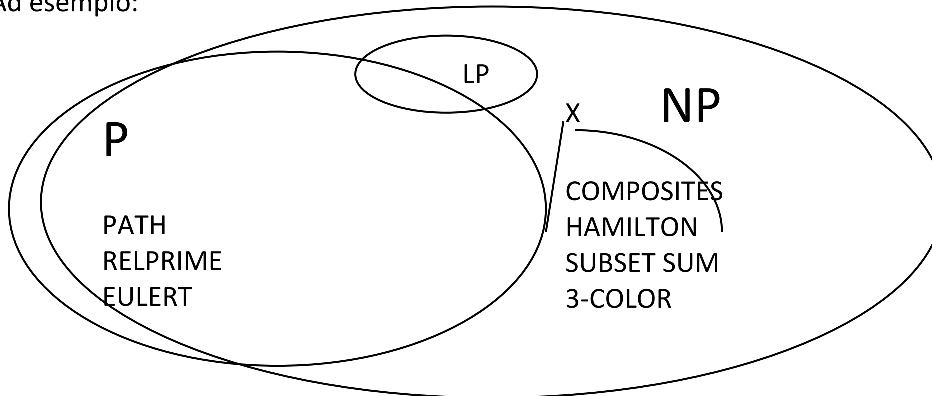
Risposta: Si

Domino[2] è un problema che appartiene a P?

Risposta: No

NP-Completezza

Pensiamo di prendere, vista la relazione di P e NP, se tramite riduzione, per esempio il circuito hamiltoniano, risolvere tutti i problemi che sono in NP.
Ad esempio:



Al confine si pone NP-Hard

Si descrivono problemi molto difficili da risolvere, chiamati NP-Hard, al confine tra i problemi facili e quelli difficili da risolvere; questi problemi possono essere risolti tramite approssimazione oppure dando dei casi particolari per cui il problema si risolve. In particolare, se esistesse un algoritmo in tempo polinomiale in grado di risolvere tutta una particolare classe di problemi, tutti quelli in NP sarebbero polinomialmente risolvibili. Questi problemi sono definiti come *NP-completi*.
Tutti i formalismi di calcolo ragionevoli sono equivalenti a meno di fattori polinomiali nei tempi di calcolo.
Esempi: Macchine di Turing Deterministiche, Linguaggi di programmazione concreti: (Java, C++, Python)
Eccezioni: Computer quantistici, DNA Computing, Bio Computing

Riassumendo:

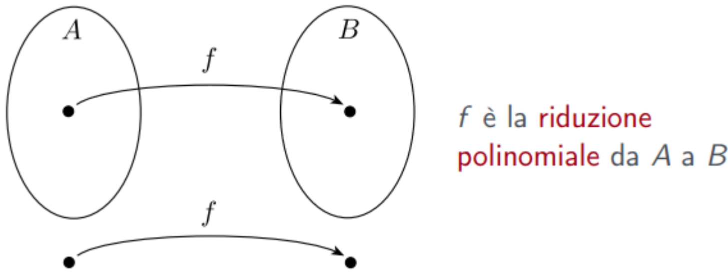
- **P** è la classe dei linguaggi in cui l'appartenenza di una stringa $x \in \Sigma^*$ al linguaggio può essere decisa da una macchina di Turing deterministica in tempo $O(|x|^k)$
- **NP** è la classe dei linguaggi in cui l'appartenenza di una stringa $x \in \Sigma^*$ al linguaggio può essere verificata da un verificatore in tempo $O(|x|^k)$.
- **Equivalente:** è la classe dei linguaggi in cui l'appartenenza di una stringa $x \in \Sigma^*$ al linguaggio può essere decisa da una macchina di Turing nondeterministica in tempo $O(|x|^k)$.
- **coNP** è la classe dei linguaggi tali che il loro complementare è in *NP*

Descriviamo la *riducibilità in tempo polinomiale*, similmente alle funzioni di riduzione già affrontate:

Definition

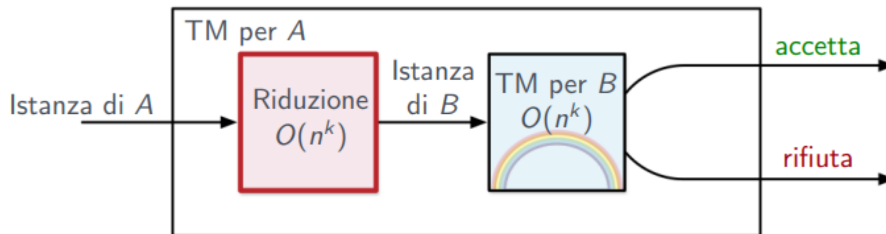
Un linguaggio A è riducibile in tempo polinomiale al linguaggio B ($A \leq_P B$), se esiste una funzione calcolabile in tempo polinomiale $f : \Sigma^* \mapsto \Sigma^*$ tale che

$$\text{per ogni } w \in \Sigma^* : w \in A \text{ se e solo se } f(w) \in B$$

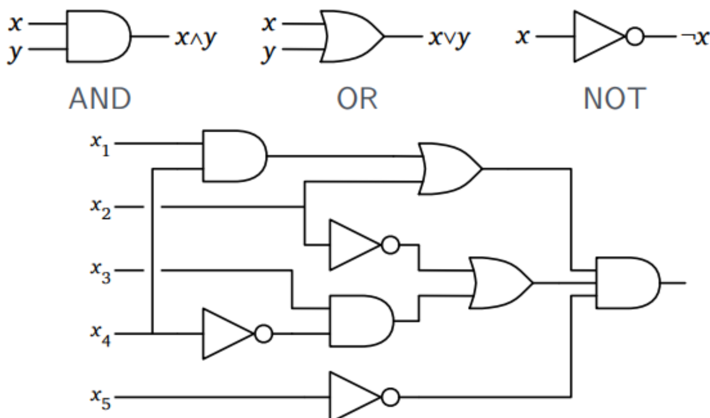


In tempo polinomiale, avremo quindi una risposta, mettendo insieme soluzioni ottenute in tempo polinomiale e dunque rimane in PTIME.

Se $A \leq_P B$, e $B \in P$, allora $A \in P$:



Discutiamo il Re dei Problemi NP (proveniente dal pdf *NP Hardness* presente su Moodle), prendendo l'esempio delle porte logiche.



Il problema è la soddisfacibilità del circuito, con queste caratteristiche:

CircuitSAT: dato un circuito Booleano, esistono dei valori di input che permettono di ottenere output $= 1$?

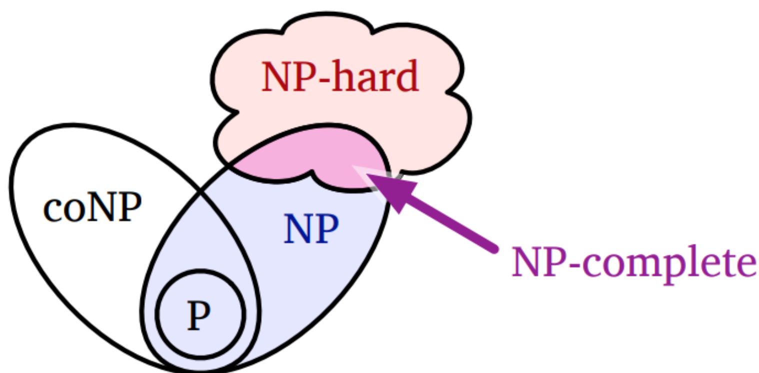
Il certificato di appartenenza è dato dall'insieme dei valori di input x_1, x_2, ...
L'operazione di verifica viene fatta in tempo polinomiale per le *n* porte logiche, calcolando l'output livello per livello, partendo dall'assegnazione del certificato (ad esempio x1 = 1, x2 = 0, x3 = 0, x4 = 1, x5 = 0); calcola gli output delle porte di primo livello, poi quelle di secondo livello, poi quelle di terzo livello… infine, calcola l'output dell'interno circuito; se è uguale ad 1, allora *SI* altrimenti *NO*.
CircuitSAT è il RE dei problemi in NP:

- Ogni problema che sta in NP può essere trasformato in una istanza di CircuitSAT in tempo polinomiale
- CircuitSAT può essere usato per risolvere tutti i problemi che stanno in NP.
- Chi scopre un algoritmo polinomiale per CircuitSAT sa risolvere tutti i problemi NP in tempo polinomiale.

L'esistenza di una macchina di Turing polinomiale per risolvere CircuitSAT implica che P = NP (Teorema di Cook e Levin, 1973).
Definiamo i problemi NP-hard:

- Un problema è NP-hard se l'esistenza di un algoritmo polinomiale per risolverlo implica l'esistenza di un algoritmo polinomiale per ogni problema in NP.
- Se siamo in grado di risolvere un problema NP-hard in modo efficiente, allora possiamo risolvere in modo efficiente ogni problema di cui possiamo verificare facilmente una soluzione, usando la soluzione del problema NP-hard come sottoprocedura.
- Un problema è NP-completo se è sia NP-hard che appartenente alla classe NP (o "NP-easy").
  - Esempio: **CircuitSAT**!

A noi interessano i problemi NP-completi:



La NP-Completezza permette di capire se il problema è intrattabile e capire se, in determinati casi, può essere trattabile. Progettare ed implementare algoritmi in questo modo richiede una conoscenza dei principi della complessità.

Spesso per risolvere un problema si adotta un approccio diverso al cercare un algoritmo efficiente per il caso generale, infatti:

- si identifica un caso particolare trattabile
- si cerca una soluzione approssimata

Le *dimostrazioni di NP-completezza* sono composte da:

Ogni dimostrazione di NP-completezza di compone di due parti:

1. dimostrare che il problema appartiene alla classe NP;
2. dimostrare che il problema è NP-hard.

- Dimostrare che un problema è in NP vuol dire dimostrare l'esistenza di un verificatore polinomiale.
- Le tecniche che si usano per dimostrare che un problema è NP-hard sono fondamentalmente diverse.

Per dimostrare che un certo problema è NP-hard si procede con *dimostrazione per riduzione polinomiale*. *Per dimostrare che B è NP-hard dobbiamo ridurre un problema NP-hard a B.*

Consideriamo come partenza il problema CircuitSAT, cercando una riduzione partendo da un problema NP-Hard andando ad un problema risolvibile con riduzione.

A NP-Hard → B è NP-Hard equivalente a $X \leq_p A \leq_p B$

(indicando con $\leq_p$ una riduzione polinomiale)

Lo schema di riduzione polinomiale è così descritto:

Per dimostrare che un problema B è NP-hard:

1. Scegli un problema A che sai essere NP-hard.
2. Descrivi una riduzione polinomiale da A a B:
   - data un'istanza di A, trasformala in un'istanza di B,
   - con una funzione che opera in tempo polinomiale.
3. Dimostra che la riduzione è corretta:
   - Dimostra che la funzione trasforma istanze "buone" di A in istanze "buone" di B.
   - Dimostra che la funzione trasforma istanze "cattive" di A in istanze "cattive" di B.
     Equivalente: se la tua funzione produce un'istanza "buona" di B, allora era partita da un'istanza "buona" di A.
4. Mostra che la funzione impiega tempo polinomiale.

Prendiamo quindi il *problema della soddisfacibilità booleana*, chiamato anche *SAT*:

- una formula Booleana come

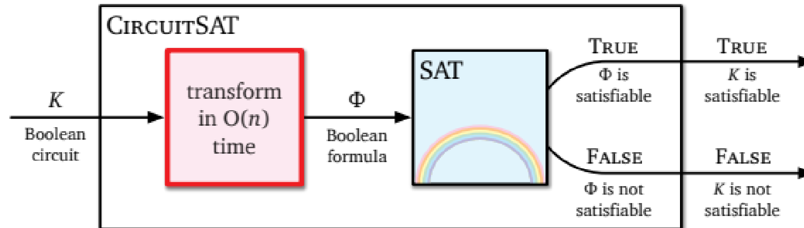
$$(a \vee b \vee c \vee \overline{d}) \leftrightarrow ((b \wedge \overline{c}) \vee \overline{(\overline{a} \rightarrow d)} \vee (c \neq a \wedge b)),$$

- è soddisfacibile se è possibile assegnare dei valori booleani (Vero/Falso) alle variabili $a, b, c, \ldots,$ in modo che il valore di verità della formula sia Vero

$$\text{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ è una formula booleana soddisfacibile}\}$$

Esso è il problema di partenza per lo studio dei problemi NP-Completi.

- SAT è in NP:
  il certificato è l'assegnamento di verità alle variabili $a, b, c, \ldots$
- SAT è NP-hard:
  dimostrazione per riduzione di **CircuitSAT** a SAT



Questa riduzione deve trasformare il circuito in una formula booleana:

1) Dare un nome agli output delle porte logiche
2) Scrivere le espressioni booleane per ogni porta logica e metterle in AND logico, aggiungendo "AND Z alla fine"

$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

Ora dobbiamo mostrare che il circuito originale K è soddisfacibile *se e solo se* la formula risultante Φ è soddisfacibile. Dimostriamo questa affermazione in due passaggi:

⇒ Dato un insieme di input che rende vero il circuito K, possiamo ottenere i valori di verità per le variabili nella formula Φ calcolando l'output di ogni porta logica di K.

⇐ Dati i valori di verità delle variabili nella formula Φ, possiamo ottenere gli input del circuito semplicemente ignorando le variabili delle porte logiche interne y_i e la variabile di uscita z.

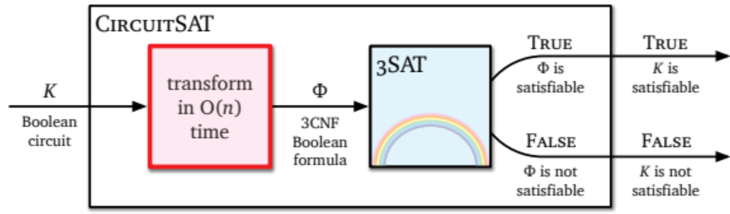
L'intera trasformazione da circuito a formula può essere eseguita in *tempo lineare*.
Inoltre, la dimensione della formula risultante cresce *di un fattore costante* rispetto a qualsiasi ragionevole rappresentazione del circuito. Per ottenere un assegnamento delle variabili che sono nell'input si ha:
K → Φ
<K> ∈ C.SAT
$X_1, \ldots, X_N, Y_1, \ldots, Y_N, Z \leftarrow 1$ (solo per Z vale 1)

- Intendiamo dimostrare l'NP-completezza di un'ampia gamma di problemi
- Dovremmo procedere per riduzione polinomiale da **SAT** al problema in esame
- Esiste però un importante problema "intermedio", detto **3SAT**, molto più facile da ridurre ai problemi tipici rispetto a **SAT**:
  - anche **3SAT** è un problema di soddisfacibilità di formule booleane
  - **3SAT** però richiede che le formule siano di una forma ben precisa, formate cioè da congiunzione logica di clausole ognuna delle quali è disgiunzione logica di tre variabili (anche negate)

Discutiamo poi il problema 3SAT, booleana in 3-CNF (CNF - Conjunctive Normal Form):

3SAT = $\{\langle\varphi\rangle \mid \varphi$ è una formula booleana in 3-CNF soddisfacibile$\}$

- 3SAT è in NP:
  il certificato è l'assegnamento di verità alle variabili $a, b, c, \ldots$
- 3SAT è NP-hard:
  dimostrazione per riduzione di **CircuitSAT** a **3SAT**



La riduzione opera con:

1. Fai in modo che ogni porta logica abbia al massimo due input
2. Trasforma il circuito in una formula booleana come fatto per **SAT**
3. Trasforma la formula in CNF usando le regole seguenti:

$$a = b \wedge c \mapsto (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$$
$$a = b \vee c \mapsto (\overline{a} \vee b \vee c) \wedge (a \vee \overline{b}) \wedge (a \vee \overline{c})$$
$$a = \overline{b} \mapsto (a \vee b) \wedge (\overline{a} \vee \overline{b})$$

4. Trasforma la formula in 3CNF aggiungendo variabili alle clausole con meno di tre letterali:

$$a \vee b \mapsto (a \vee b \vee x) \wedge (a \vee b \vee \overline{x})$$
$$a \mapsto (a \vee x \vee y) \wedge (a \vee \overline{x} \vee y) \wedge (a \vee x \vee \overline{y}) \wedge (a \vee \overline{x} \vee \overline{y})$$

Passare da 3SAT a 2SAT si ottiene un problema polinomiale (cosa comune che, passando da 3 a 2, spesso si risolve in tempo polinomiale) (slide 62 del set di slide *20 – npcomplete.pdf*)

Algoritmo polinomiale per **2SAT**
(soddisfacibilità di formule in 2CNF):

- Prendiamo una variabile x e assegnamo valore 1 (vero)
- In ogni clausola con $\overline{x}$, l'altro letterale deve essere vero
  - **Esempio:** in $(\overline{x} \vee \overline{y})$, y deve essere falso (0)
- Continuiamo assegnado le variabili il cui valore è "forzato"

L'algoritmo assegna qui valori alle variabili finché non succede una di tre cose:

1) una contraddizione: una variabile è forzata ad essere sia vera che falsa
2) tutte le variabili con valore forzato sono state assegnate, ma ancora ci sono clausole non soddisfatte
3) si ottiene un assegnamento che dà valore vero alla formula

Di conseguenza:

1) Ci può essere un assegnamento che dà valore vero solo valore falso per la variabile x. Ricominciamo assegnando x a 0 (falso).
2) Ci sono ancora variabili e clausole che non sono assegnate. Eliminiamo le clausole soddisfatte, e ripartiamo.
3) La risposta è Si (ho trovato un assegnamento che soddisfa la formula).

<u>Domande Wooclap</u>

Associa ogni classe di complessità alla sua definzione

P

A. linguaggi che si possono decidere in tempo polinomiale

NP

B. linguaggi che si possono verificare in tempo polinomiale

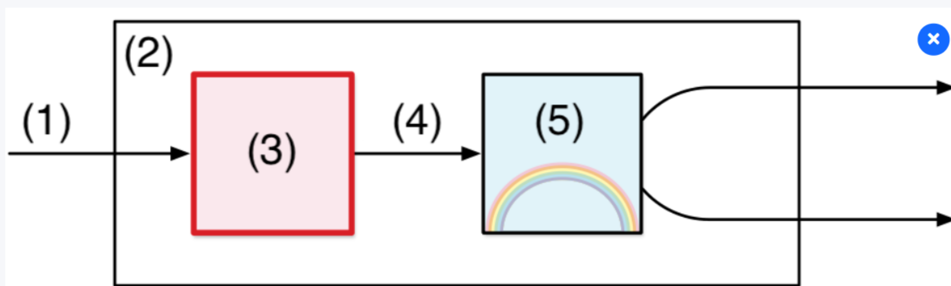
L'esistenza di un algoritmo polinomiale per un problema NP-Hard implica l'esistenza di un algoritmo polinomiale per ogni problema di complessità:

Risposta: NP

Un problema NP-completo è un problema che è contemporaneamente ..... e ......
Seleziona due tra le scelte possibili:

Risposte:

- in NP
- in NP-Hard



Completa lo schema di riduzione che dimostra che B è un problema NP-Hard usando A come problema NP-Hard di riferimento. Associa ogni etichetta alla sua posizione nella figura.

Risposte:

1) Istanza di A
2) TM per A
3) Riduzione polinomiale
4) Istanza di B
5) TM per B

Scritto da Gabriel

Disponi i passi della dimostrazione che SAT è NP-Hard:

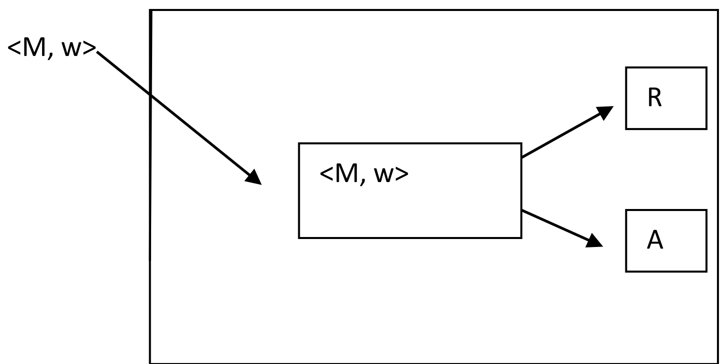
1) Scelgo CircuitSAT come problema che so essere NP-Hard
2) Descrivo una funzione di riduzione polinomiale da CircuitSAT a SAT
3) Dimostro che la funzione di riduzione è corretta
4) Dimostro che la funzione di riduzione impiega tempo polinomiale
5) Siccome CircuitSAT ≤P SAT e CircuitSAT è NP-Hard, allora SAT è NP-Hard

<u>Tutorato 7</u>

A_{TM} = {<M, w> | M è una TM e M accetta w}

$HALT_{TM}$ = {<M, w> | M è una TM, M si blocca su input w }

R è decisore $HALT_{TM}$



La riduzione è una funzione che prende in input la funzione di partenza e, applicata si A e B
F : $\Sigma^* \rightarrow \Sigma^*$
w $\mapsto \Sigma^*$

Per ogni w che appartiene ad A ←→ f(w) appartiene a B
sse

Se A è riducibile a B B è decidibile e dunque A è decidibile

Se chiedono di dimostrare che qualcosa è indecidibile

Avendo A che è riducibile a B A è indecidibile allora B è indecidibile
Se esiste una f che è riduzione:

F in input <M, w> e vado a costruire M' sull'input x
Se M accetta, *accetta*
Se M rifiuta, *rifiuta*

Avendo una condizione A ←→ B, si può trasformare in:
A → B w in A → f(w) in B
F(w) in B → w in A

<u>A</u> → <u>B</u> w in A → F(w) in B

Scritto da Gabriel

w not in A → F(w) not in B

Dimostriamo che F è effettivamente una riduzione da A_{TM} ad $HALT_{TM}$.
<M, w> appartiene a A_{TM} → <M, w'> appartiene a $HALT_{TM}$
<M, w> non appartiene a A_{TM} → <M, w'> non appartiene a $HALT_{TM}$
A_{TM} ={<M, w> | M TM M accetta w}
$HALT_{TM}$ = {<M, w> | M è una TM e M si ferma}

A noi che accetti o meno non interessa, basta che *si fermi*.
La seguente è la definizione di riduzione:
<M, w> appartiene a A_{TM} ←→ F(<M, w>) appartiene a $HALT_{TM}$

In particolare, diciamo che su F in input <M, w>, costruisco M'
M' sull'input x:
- x gira su M
- se M accetta M', *accetta*
- se M rifiuta M', *loop*

Sulle simbologie, quando si ha il loop, anche M' andrà in loop e non ci saranno problemi.

Se avessimo una cosa che si ferma sempre, eviteremmo facilmente il loop.
La riduzione è già di per sé una dimostrazione per assurdo.
Infatti, immaginiamo per assurdo che R sia decisore $HALT_{TM}$; essa ci semplifica la vita dato che possiamo fare una dimostrazione più meccanica e comunque ottenere una risposta.

F: Input <M, w>
Mw = Su input x
1. se x ≠ 1010, rifiuta
2. se x = 1010, esegue
3. se M accetta (quindi è l'output), *accetta*
4. se M rifiuta, *rifiuta*

OUTPUT: <M, w>

GOAL
1) <M, w> appartiene a A_{TM} → F(M, w) = Mw appartiene a A_{1010}
2) <M, w> non appartiene ad A_{TM} → F(M,w) = Mw non appartiene a A_{1010}

Caso (1): Mw appartiene ad A_{TM} → Accetta
Caso (2): Mw non appartiene ad A_{TM} → Rifiuta o va in loop

<u>Tutorato 8</u>

L_2 = { <M, w>|M è una TM e accetta ww^r }

La prima strategia è di dimostrare che esiste, per contraddizione, una TM R che è un decisore di L_2
La strategia è più complicata rispetto invece ad avere:
$A_{TM} <=_m L_2$
La funzione di riduzione F prende in input un'istanza del mio problema nel seguente modo.
F <M, w> = <M*, w*>

Partendo quindi da un'istanza <M, w> che appartiene ad A_{TM} → F <M, w> appartiene a L_2
Quindi:
(1) <M, w> appartiene a A_{TM} → F <M, w> appartiene a L_2
(2) <M, w> non appartiene a A_{TM} → F <M, w> non appartiene a L_2

Scritto da Gabriel

Automi semplici (per davvero)

L'obiettivo è di verificare cosa la fa la TM sull'input x e fare in modo accetti per certo le condizioni del linguaggio; se accetta la stringa, la macchina esegue e se non accetta la stringa che gli do in input l'automa rifiuta. Se va in loop in automatico M va in loop, altrimenti accetta.
Noi lo facciamo non tanto per costruire una TM che rispetti il linguaggio ma devono valere le condizioni (1) e (2). In questo modo dimostriamo che anche L è indecidibile
F su input <M, w>

1) Costruire M_R
   M_R = su input x:
   1. $x = ww^R$, *accetta*
   2. Se $w = w^{R'}$:
   3. Se M accetta w, *accetta*
   4. Se M rifiuta w, *rifiuta*
2) Output <M_R, w>

Se la stringa non appartiene, date le condizioni precedenti, se rifiuta M_R rifiuta, allora la stringa non appartiene. La logica è la stessa per qualsiasi stringa; se diamo un input diverso da quello che ho in ingresso, allora la riduzione rifiuta sempre.
Pertanto, la riduzione è potente.
Le condizioni più importanti sono 2/3/4, in funzione della 1.
Quindi esegue *alternativamente* i punti 2/3/4.

Discussione di NP-Hard

NP-Complete si pone in mezzo a NP ed NP-Hard.
I problemi NP-Hard si risolvono per riduzione, partendo dal problema che si sa sia NP-Hard dal problema che si ha e si dimostra che sia NP-Completo.
Il certificato del problema deve essere valido in tempo polinomiale.

Idea: A è NP-Completo

1) Dato un certificato il controllo è polinomiale
2) Dato x NP-Hard noto $X <=_p A$

1) Riduzione Polinomiale di una generica istanza X in una specifica di A
Non ci interessa ridurre le istanze da A ad x, voglio ridurli solo in senso che va da x ad A.

Buona istanza === Certificato
Ora noi logicamente ragioniamo che B → A

2) Dato un certificato valido di X ottengo un certificato valido di A

3) Dato un certificato valido di A ottengo un certificato valido di X

La (2) e la (3) non sono uguali e tutto ciò è legato al punto 3.
Nel punto 2 prendo un generico certificato per X e dimostro che è valido per A.
Per 3 si ha una riduzione da A ad X.
Noi prendiamo per A un certificato che prima è stato ridotto

PebbleDestruction

Noi sappiamo che abbiamo un grafo in cui si ha un certificato che è composto da un cammino Hamiltoniano su un grafo; dato il certificato, il cammino è presente.

Scritto da Gabriel

Tolgo due ciottoli da un nodo e li inserisco in quello adiacente. Il gioco termina quando mi ritrovo un solo nodo che termina con un singolo ciottolo.
Ricorsivamente applico le regole:

- Tolgo 2 e ne metto 1 ad uno adiacente
- Ad un certo punto arrivo con un vertice ad un sassolino solo

Io so per ipotesi che il percorso hamiltoniano esiste; se "torno indietro" rimetto i sassolini come prima.
Prendo un vertice, gli assegno 2, proseguo nel cammino associando a tutti il valore 1 e all'ultimo associo valore 0. Tutto è eseguito in tempo lineare e dunque è polinomiale.

Facciamo il ragionamento sbagliato:

- Ipotizziamo che noi partiamo da un'istanza di Pebble che considera. Applichiamo la mossa e arrivo ad un certo punto in cui non ho percorso tutti i vertici e rimango con un sassolino. L'ordine delle mosse è indicato dagli archi.

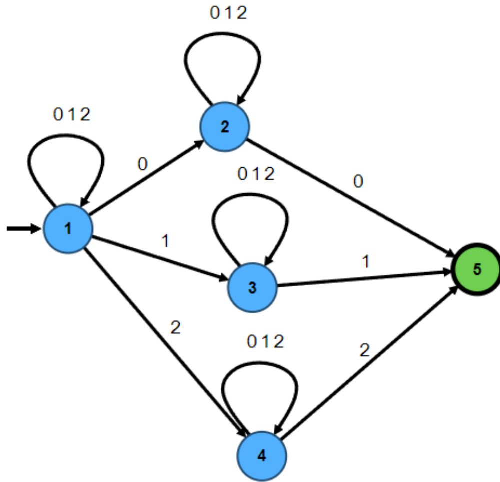
L'istanza che dobbiamo considerare è l'istanza del problema di partenza che viene trasformata in un certificato valido. Nel momento in cui ho un circuito hamiltoniano e lo trasformo (prendendo un vertice con 2, e altri due vertici con 1 solo sassolino).
Tramite l'istanza valida, noi torniamo indietro avendo un circuito hamiltoniano.
La trasformazione è data da:
Dato un certificato valido di X ottengo un certificato valido di A

La dimostrazione è realizzata solo sulle istanze valide.
L'obiettivo è che dato un certificato il contratto è polinomiale e dato X un numero noto, si opera una riduzione polinomiale attraverso A.

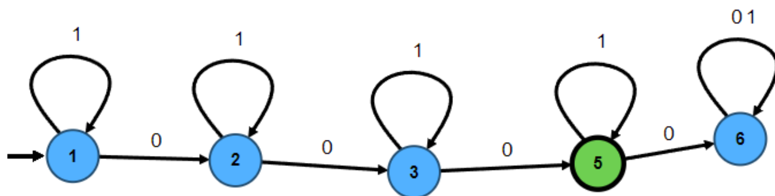
Esercizi NFA/ε-NFA a DFA/NFA a DFA

Primo set di esercizi Automata Tutor

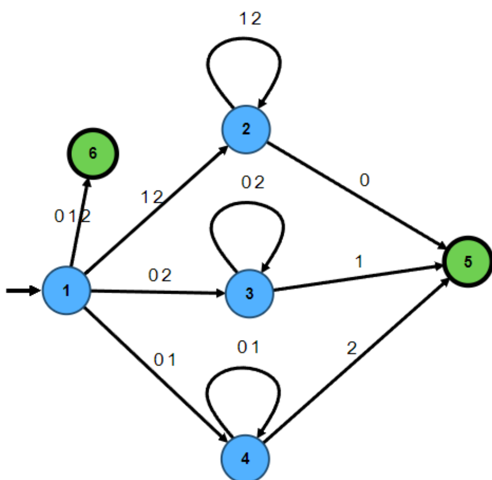
1) *Automa NFA con alfabeto {0,1,2} che ha come linguaggio:*
   la cifra finale sia comparsa in precedenza



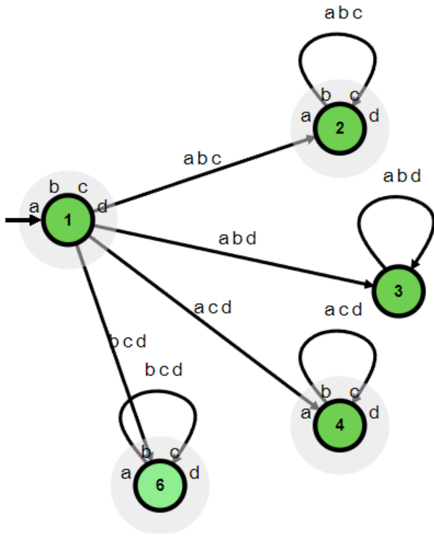
2) *Automa DFA con alfabeto {0, 1} che ha come linguaggio:*
   tutte e sole le stringhe che contengono esattamente tre zeri (anche non consecutivi)



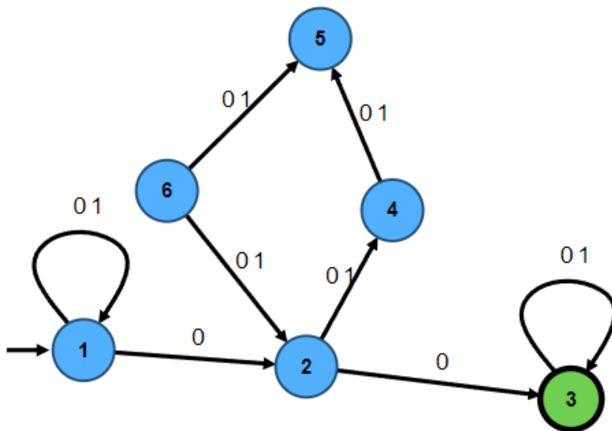
3) *Automa NFA con alfabeto {0,1,2} che ha come linguaggio le stringhe in cui:*
   la cifra finale non sia comparsa in precedenza



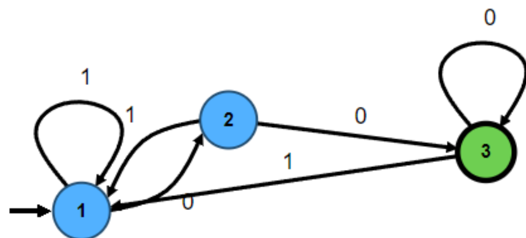
4) *Automa NFA con alfabeto {a, b, c, d} che ha come linguaggio le stringhe in cui: uno dei simboli dell'alfabeto non compare mai*



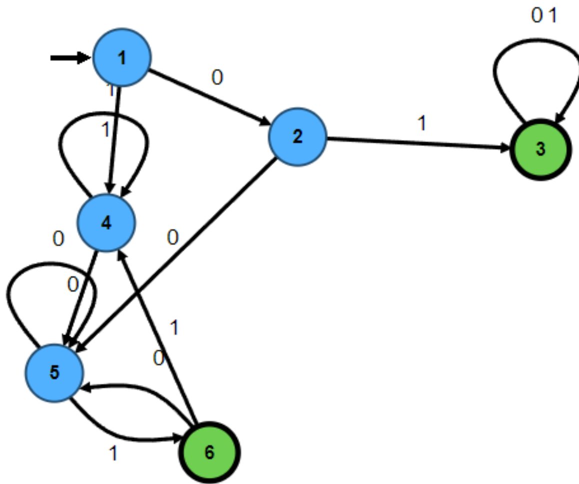
5) *Automa NFA con alfabeto {0, 1} che ha come linguaggio le stringhe in cui: esistono due 0 separati da un numero di posizioni multiplo di 4*



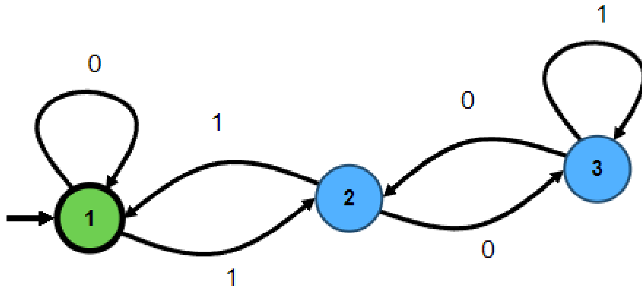
6) *Automa DFA con alfabeto {0, 1} che ha come linguaggio: tutte e sole le stringhe che terminano con 00*



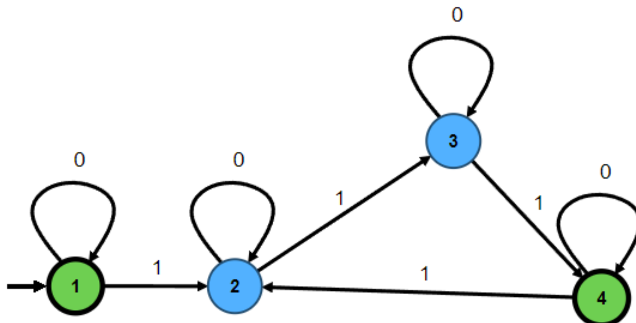
7) *Automa DFA con alfabeto {0, 1} che ha come linguaggio:*
tutte e sole le stringhe che cominciano o finiscono con 01 (o entrambe le cose)



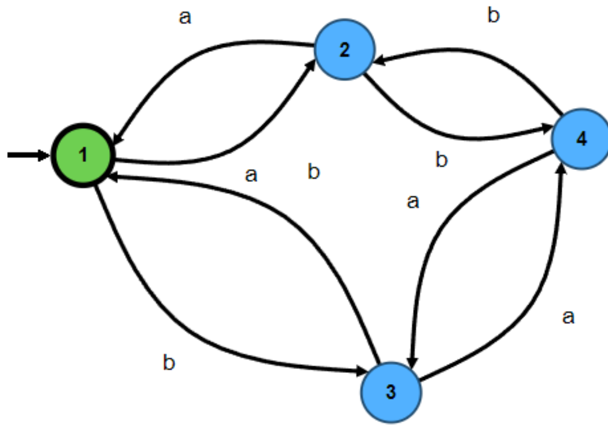
8) *Automa DFA con alfabeto {0, 1} che ha come linguaggio:*
tutte le stringhe che rappresentano la codifica binaria di un numero multiplo di 3. La stringa vuota non rappresenta nessun numero.



9) *Automa DFA con alfabeto {0, 1} che ha come linguaggio:*
tutte e sole le stringhe che un numero di 1 multiplo di 3



10) Automa DFA con alfabeto {0, 1} che ha come linguaggio: tutte e sole le stringhe con un numero pari di "a" e di "b"

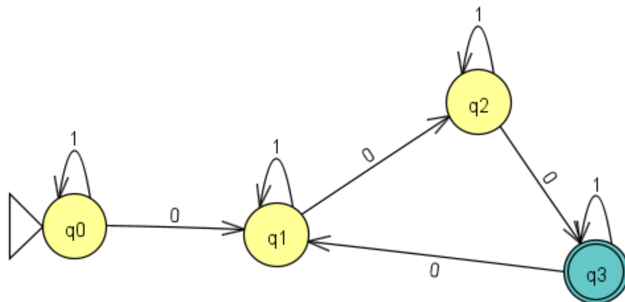


Dal file – Esercizi01 (uno dei due del file è stato fatto qui sopra)

1. Costruite un DFA che riconosce il linguaggio

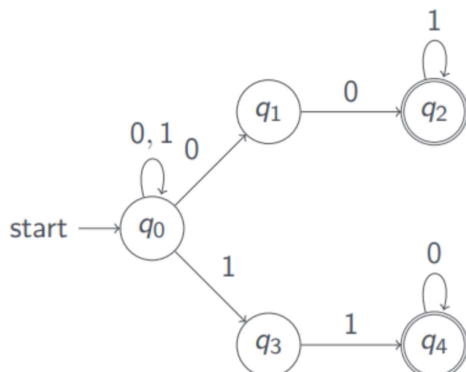
$$L_1 = \{w \in \{0,1\}^* \mid w \text{ contiene un numero di 0 multiplo di 3}\}$$

Per esempio, 000, 00110 e 010101010101 appartengono al linguaggio perché contengono rispettivamente 3, 3 e 6 zeri, mentre 00, 001010 e 0101010101, che contengono 2, 4 e 5 zeri, non appartengono al linguaggio.



Conversioni da NFA a DFA (esercizi delle slide)

Trasformare il seguente NFA in DFA



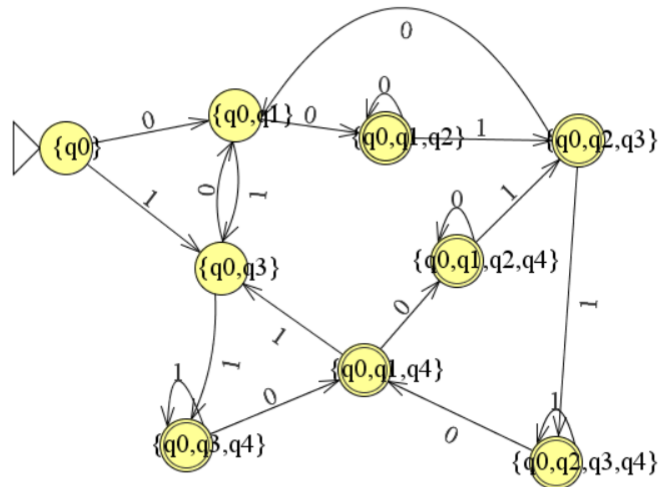
Per poterlo fare:

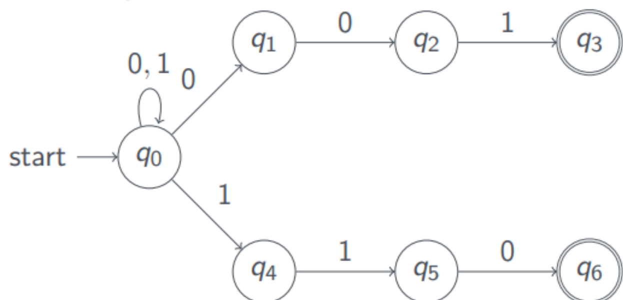
5) prima si costruisce la tabella di transizione
6) successivamente si costruisce l'automa. Si parte dallo stato iniziale (q0) e poi si va avanti ragionando per unione. Descrivo esattamente cosa si fa:
1) parti da q0 (stato iniziale); per 0 vai verso (q0,q1) mentre per 1 vai verso (q0,q1).
2) Ora dobbiamo guardare la tabella. Siamo in (qo,q1); ciò implica che dovremo andare avanti guardando l'unione delle celle corrispondenti nella tabella (quindi farò l'unione di q0 e q1). Questo significa che per 0 andrò verso (q0,q1,q2) mentre per 1 andrò verso (q0,q3).
3) In questo modo si procede per ogni singolo stato. Guardo dove sono e unisco (come si vede dai colori) ciò che ci sta nella tabella e converto correttamente.

   Una considerazione; si procede per unione come detto e ogni singolo stato sia indicato come finale, anche l'unione sarà disegnata come finale (ad esempio (q0,q1,q2) oppure (q0,q2,q3) sono stati finali).

Q_d	0	1
∅	∅	∅
->{q0}	{q0, q1}	{q0, q3}
{q1}	{q2}	∅
*{q2}	∅	{q2}
{q3}	∅	{q4}
*{q4}	{q4}	∅

Per ricavarsi tutti gli altri stati si consideri il ragionamento scritto. Poi si ricava (con pazienza e andando con calma) tutto il resto in maniera abbastanza facile.



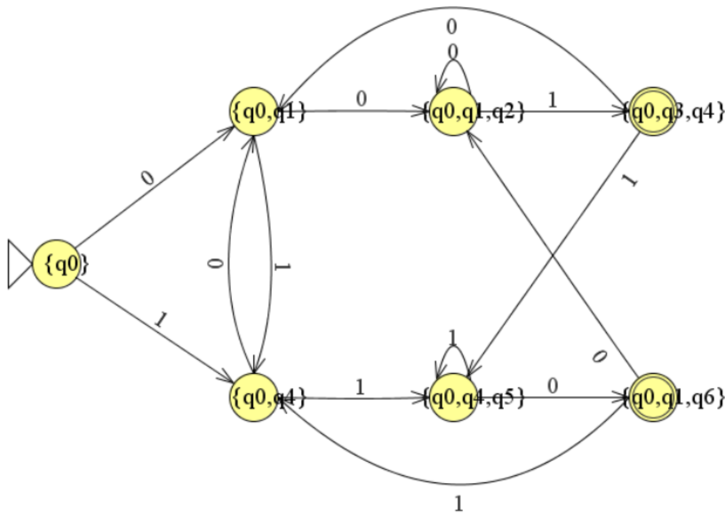


1 determinare il linguaggio riconosciuto dall'automa
2 costruire un DFA equivalente

1) L'automa riconosce come linguaggio:
Un alfabeto che comprende delle stringhe che terminano con 001 oppure con 110.
2)

Q_d	0	1
Ø	Ø	Ø
{q0}	{q0, q1}	{q0,q4}
{q1}	{q2}	Ø
{q2}	Ø	{q3}
*{q3}	Ø	Ø
{q4}	Ø	{q5}
{q5}	{q6}	Ø
*{q6}	Ø	Ø

Si procede nello stesso modo descritto brevemente sopra:



Tratto dal file 03-esercizi e visto in classe:

Operazioni su linguaggi

Siano L e M due linguaggi regolari. Definiamo le operazioni regolari di *unione*, *concatenazione* e *star* come segue:

- **Unione:** $$L \cup M = \{w \mid w \in L \text{ oppure } w \in M\}$$
- **Concatenazione:** $$L.M = \{uv \mid u \in L \text{ e } v \in M\}$$
- **Star:** $$L^* = \{x_1x_2x_3 \dots x_n \mid n \geq 0 \text{ e ogni } x_i \in L\}$$

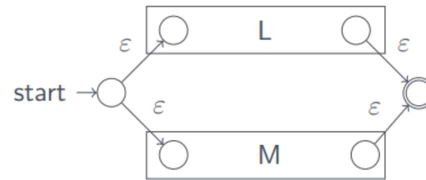
Supponi di avere a disposizione gli ε-NFA A_L e A_M che riconoscono i linguaggi L e M. Rispondi alle seguenti domande:

1. Esiste un ε-NFA che riconosce $L \cup M$? E $L.M$? E L^*?
2. Se si, descrivi degli algoritmi per costruire questi ε-NFA.

Automi semplici (per davvero)

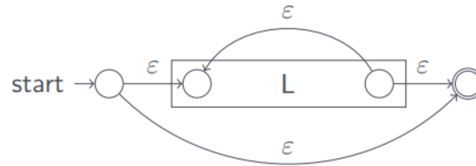
1) Sì a tutte e 3 le domande
2) Intuitivamente (poi seguono gli automi di riferimento):
3) per l'unione basta avere uno stato iniziale comune ed una biforcazione verso due stati
4) per la concatenazione si avrà uno stato iniziale seguito da uno stato finale oppure uno non finale
5) per lo star, basta avere tutte le combinazioni da e verso altri stati

■ **L + M**



■ **L.M**

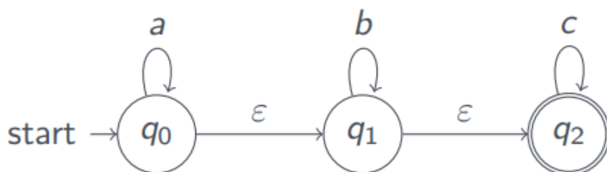
■ **L***



Esercizio con ε-chiusure (da slide 02-nfa)

1. Costruiamo un ε-NFA che riconosce le parole costituite da
   - zero o più *a*
   - seguite da zero o più *b*
   - seguite da zero o più *c*
2. Calcolare ECLOSE di ogni stato dell'automa
3. Convertire l'ε-NFA in DFA

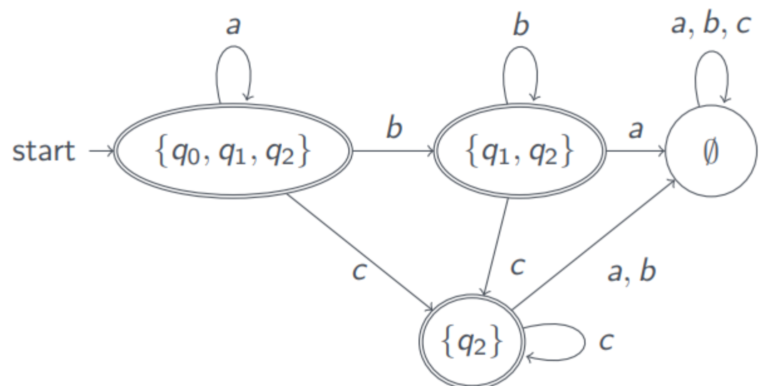
1)



2)

$$\mathrm{ECLOSE}(q_0) = \{q_0, q_1, q_2\}$$
$$\mathrm{ECLOSE}(q_1) = \{q_1, q_2\}$$
$$\mathrm{ECLOSE}(q_2) = \{q_2\}$$

3) Si ottiene l'automa anche qui applicando le regole viste per conversione NFA/DFA, considerando che le epsilon sono transizioni vuote (quindi si vede che (q0,q1,q2) può essere rappresentata come unione per ε e si ragiona anche qui appunto per unione. Ad esempio (caso stato iniziale dal calcolo delle chiusure, quindi (q0,q1,q2) confrontando con l'automa e ragionando per unione alla NFA/DFA):



6) per (a) si vede che l'unione tra q0,q1,q2 porta (q0,q1,q2), sempre perché ε è vuoto e permette di andare verso q1,q2 senza consumare nulla.

Scritto da Gabriel

7) per (b) si vede da sopra che, va verso q1. Essendo poi tra q1 e q2 una ε è transizione vuota, quindi comprende anche q2. Da cui l'unione q1,q2.
8) per (c) si vede che va solo verso q2

Ora però si ragiona come qui, ma appunto considerando solo gli stati dati dalle ε-chiusure (quindi (q1,q2) e (q2), oltre allo stato vuoto, qui considerato).

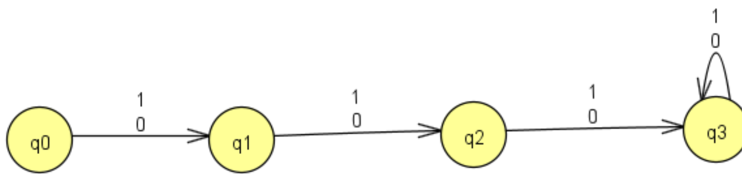
Tutorato 1

DFA

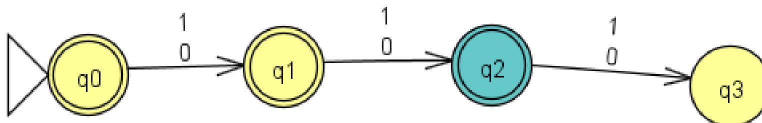
Per ognuno dei seguenti linguaggi sull'alfabeto $\Sigma = \{0, 1\}$, costruisci un DFA che accetti il linguaggio.

1. $\{w \in \Sigma^* \mid |w| = 2\}$
2. $\{w \in \Sigma^* \mid |w| \leq 2\}$
3. $\{w \in \Sigma^* \mid |w| \mod 2 = 0\}$
4. $\{w \in \Sigma^* \mid \text{ogni } 0 \text{ è seguito da } 11\}$
5. $\{w \in \Sigma^* \mid \text{contiene } 000 \text{ come sottostringa }\}$
6. $\{w \in \Sigma^* \mid |w| \mod 3 = 0\}$
7. $\{w \in \Sigma^* \mid |w| \mod 3 = 1\}$

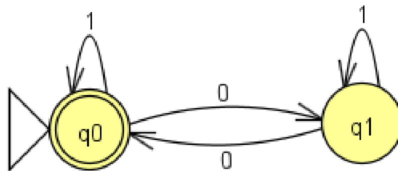
1)



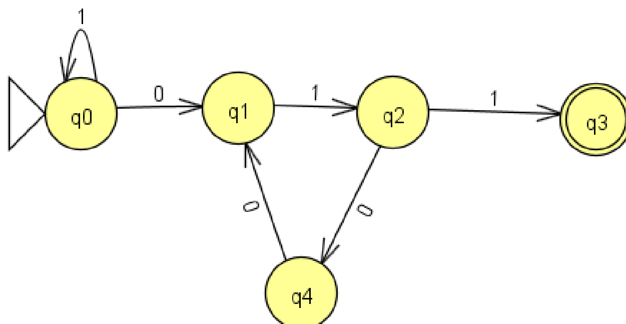
2)



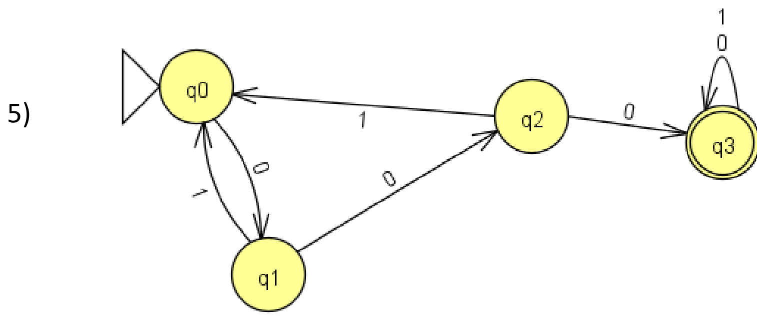
3)



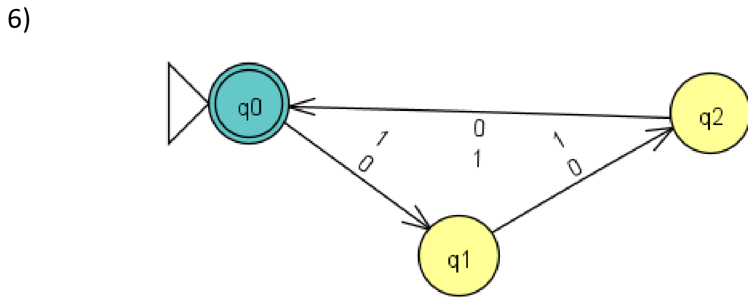
4)



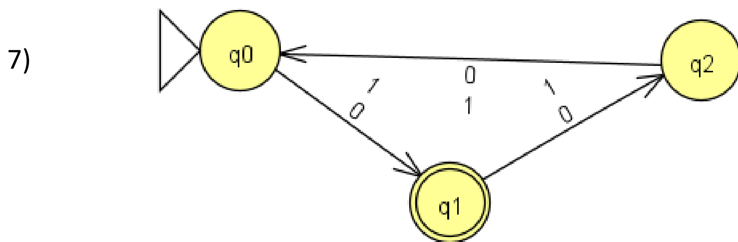
5)



6)



7)

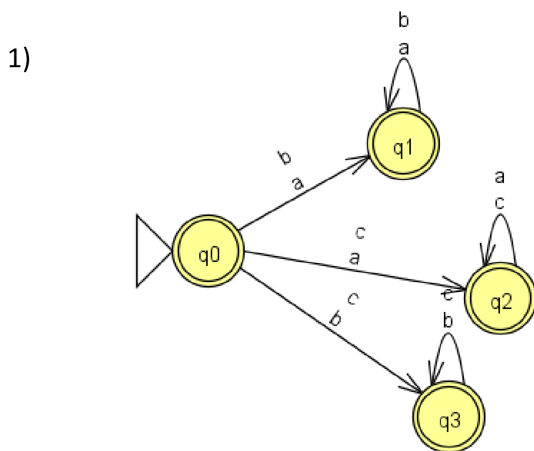


NFA e ε-NFA

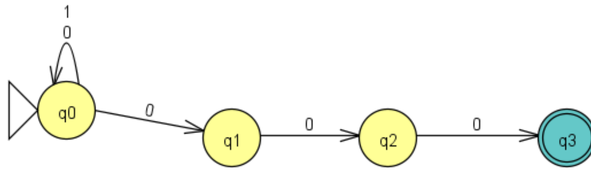
Per ognuno dei seguenti linguaggi, costruisci un ε-NFA che accetti il linguaggio.

1. $\{w \in \{a, b, c\}^* \mid$ non compaiono tutti i simboli$\}$
2. $\{w \in \{0, 1\}^* \mid$ contiene almeno tre 000 consecutivi$\}$
3. $\{w \in \{0, 1\}^* \mid$ contiene al suo interno la stringa 11 oppure 101$\}$

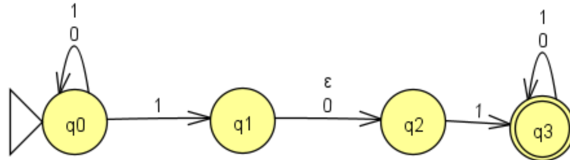
1)



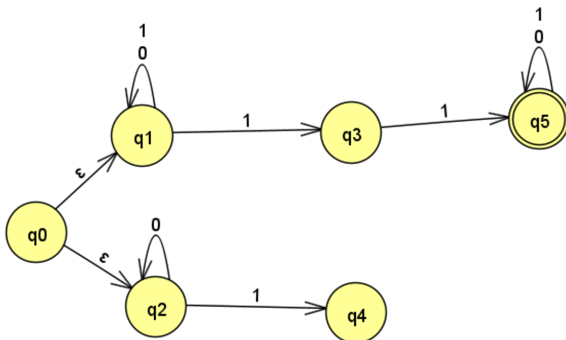
2)



3)



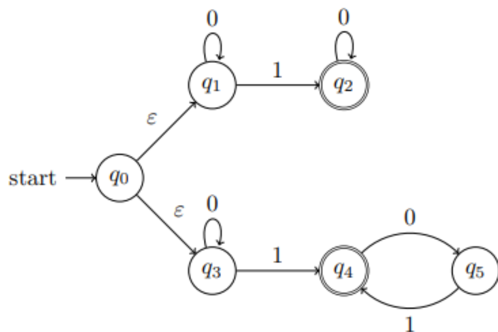
Questa sopra è la strada migliore; altrimenti l'idea può essere più lunga e una roba di questo tipo (nota: non è del tutto completa):



Conversione NFA → DFA

Trasforma ciascuno dei seguenti ε-NFA in DFA usando la costruzione per sottoinsiemi.

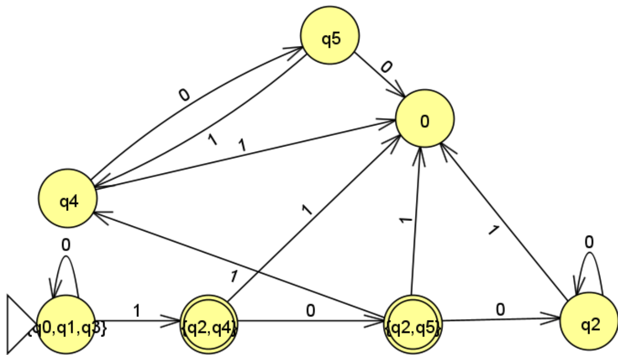
1.



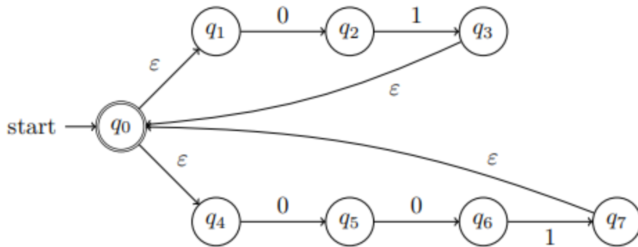
1) Primo step: calcolare le ε-chiusure
In questo caso avremmo (q0,q1) e (q0,q3). Lo stato iniziale sarà quindi (q0,q1,q3).
ENCLOSE (q0) = {q0,q1,q3}

2) Si calcola poi come sempre la tabella di transizione; sempre per unione e osservazione dello stato attuale e stati precedenti, in maniera complementare a quanto visto sopra.
Consiglio: mettere subito lo stato vuoto, perché come si vede servirà nel caso ci siano altri stati vuoti che lo raggiungono.

	0	1
∅	∅	∅
-> {q0,q1,q3}	{q0,q1,q3}	{q2,q4}
{q1,q3}	{q1,q3}	{q2,q4}
*{q2,q4}	{q2,q5}	∅
*{q2,q5}	{q2}	{q4}
{q2}	{q2}	∅
{q4}	{q5}	∅
{q5}	∅	{q4}



2.

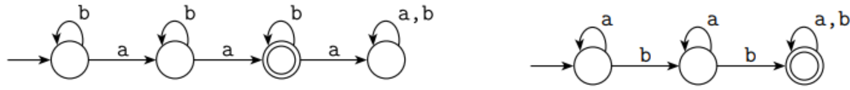


1)
ε-chiusure:
ECLOSE (q0) = {q3,q0,q1}
ECLOSE (q3) = {q3,q0,q1,q4}
ECLOSE (q7) = {q3,q0,q1,q4}

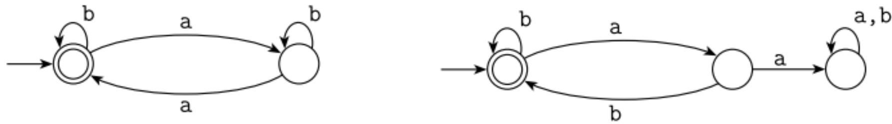
	0	1
∅	∅	∅
-> *{q3,q0,q1,q4}	{q2, q5}	∅
{q2,q5}	{q6}	{q3}
q6	∅	{q7}
q7	{q2, q5}	∅
*{q0,q1,q4}	{q2, q5}	∅

NFA/DFA Extra

1.4 **(b)** The following are DFAs for the two languages $\{w|\ w$ has exactly two a's$\}$ and $\{w|\ w$ has at least two b's$\}$.

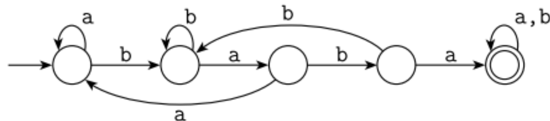


(d) These are DFAs for the two languages $\{w|\ w$ has an even number of a's$\}$ and $\{w|$ each a in w is followed by at least one b$\}$.

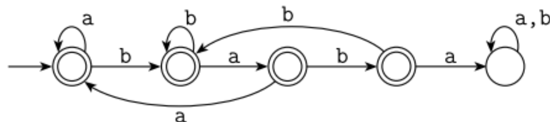


1.5 **(a)** The left-hand DFA recognizes $\{w|\ w$ contains ab$\}$. The right-hand DFA recognizes its complement, $\{w|\ w$ doesn't contain ab$\}$.

(b) This DFA recognizes $\{w|\ w$ contains baba$\}$.



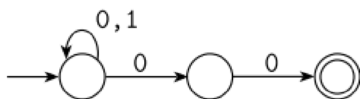
This DFA recognizes $\{w|\ w$ does not contain baba$\}$.



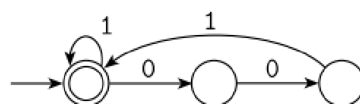
Nello screen sotto:

a) The language {w| w ends with 00} with three states
f) The language $1^* (001^+)^*$ with three states

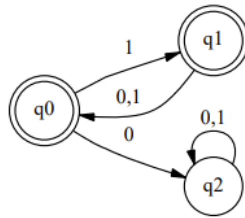
1.7 **(a)**



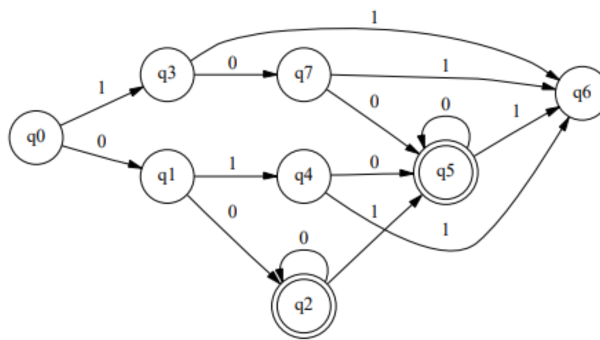
(f)



9. $\{w|$ every odd position of w is $1\}$.

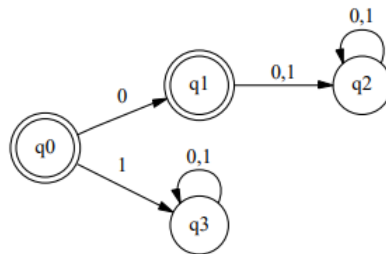


10. $\{w|w$ contains at least two 0s and at most one $1\}$.



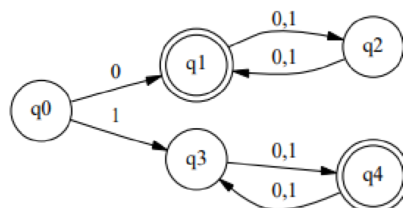
11. $\{\epsilon, 0\}$.

Qui a sinistra significa che accetta come linguaggio uno 0 e finisce ma accetta anche la stringa vuota:

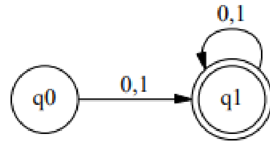


12. $\{w|w$ contains an even number of 0s, or exactly two 1s$\}$.

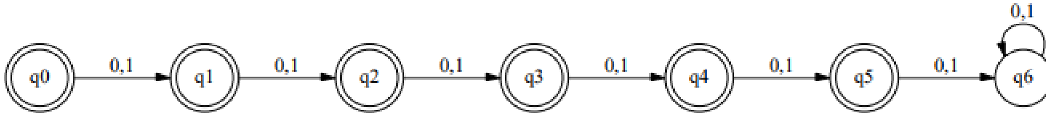
5. $\{w|w$ starts with 0 and has odd length, or starts with 1 and has even length$\}$.



14. All strings except the empty string.

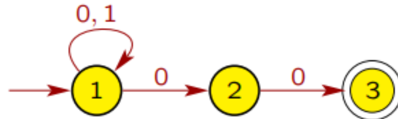


7. $\{w |$ the length of w is at most $5\}$.

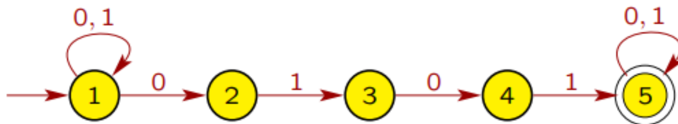


NFA Extra:

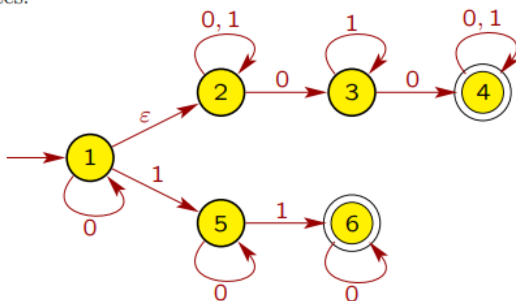
(a) The language $\{ w \in \Sigma^* \mid w \text{ ends with } 00 \}$ with three states.



(b) The language $\{ w \in \Sigma^* \mid w \text{ contains the substring } 0101\text{, i.e., } w = x0101y \text{ for some } x, y \in \Sigma^* \}$ with five states.



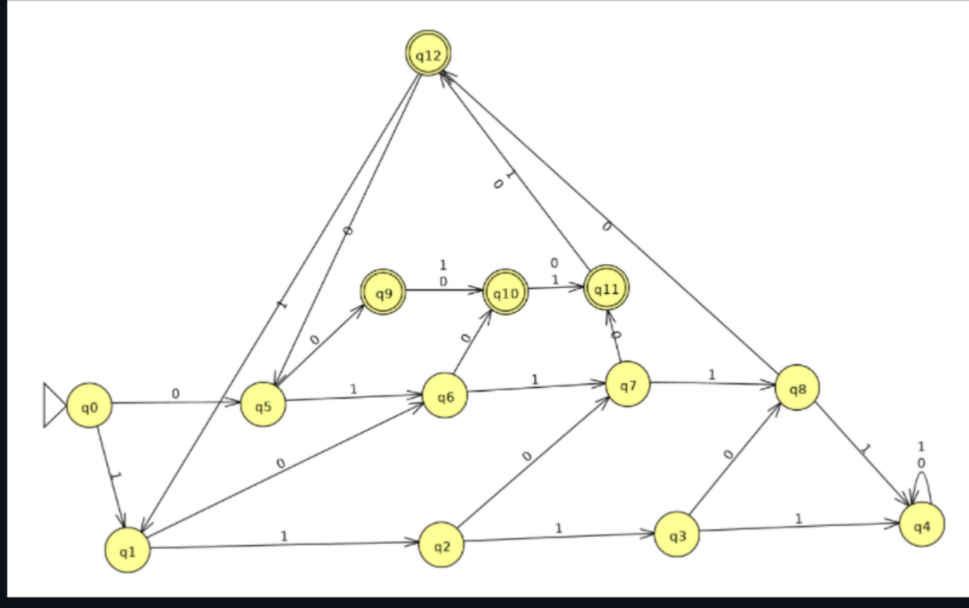
(c) The language $\{ w \in \Sigma^* \mid w \text{ contains at least two 0s, or exactly two 1s} \}$ with six states.



DFA che accetta l'insieme delle stringhe sull'alfabeto {0, 1} tali che ogni blocco di 5 simboli consecutivi contenga almeno due 0.

Ad esempio:

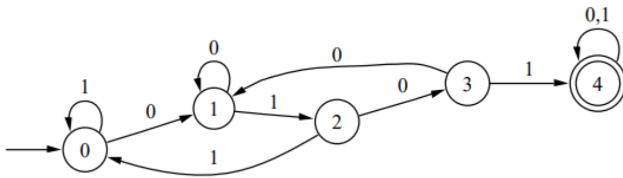
- `00011 10101` è accettato
- `10100 10110 00` è accettato
- `11011 1010` non è accettato perchè nel primo blocco manca uno 0



Construct a DFA which accepts the following language:

$$L = \{w \mid w \in \Sigma^* \wedge w \text{ contains the substring } 0101\}$$

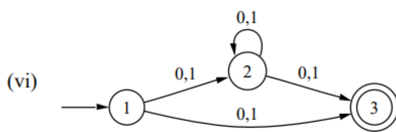
That is, $w = x0101y$ for two arbitrary strings x and y.

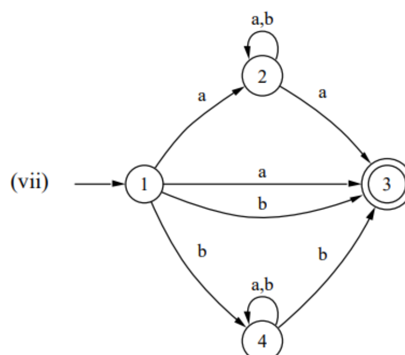


Construct finite automata, deterministic of non-deterministic for the following regular expressions.

(vi) $(0 \cup 1)(0 \cup 1)(0 \cup 1)^*$

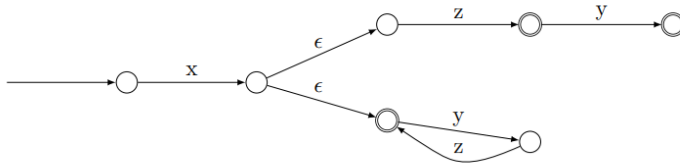
(vii) $a(a \cup b)^*a \cup b(a \cup b)^*b \cup a \cup b$





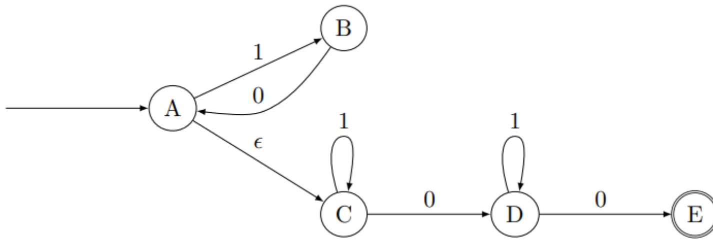
(a) The language denoted by `x(zy?|(yz)*)`.

Answer:

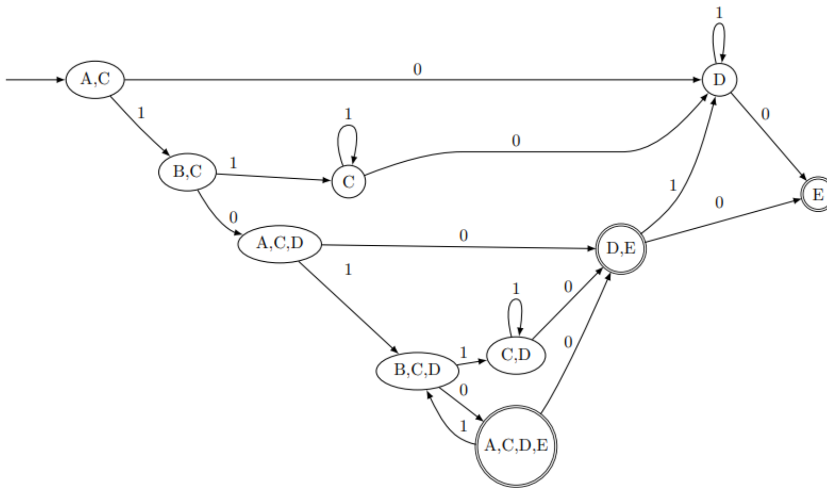


(b) The language denoted by `(10)*1*01*0`.

Answer:



Converti poi quest'ultimo in DFA:

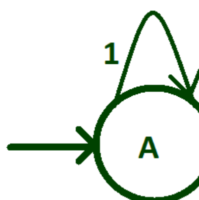


01 – Progettare DFA

1. Costruite un DFA che riconosce il linguaggio

$$L_1 = \{w \in \{0,1\}^* \mid w \text{ contiene un numero di 0 multiplo di 3}\}$$

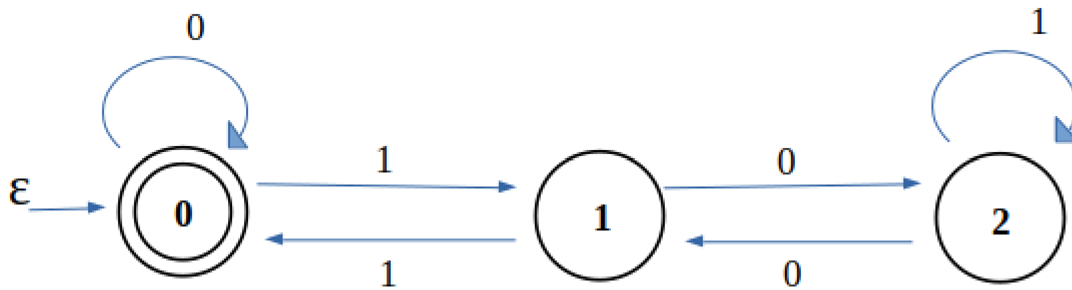
Per esempio, 000, 00110 e 010101010101 appartengono al linguaggio perché contengono rispettivamente 3, 3 e 6 zeri, mentre 00, 001010 e 0101010101, che contengono 2, 4 e 5 zeri, non appartengono al linguaggio.



2. Costruite un DFA che riconosce il linguaggio

$$L_2 = \{w \in \{0,1\}^* \mid w \text{ è la codifica binaria di un mumero multiplo di 3}\}$$

Per esempio, 11, 110 e 1001 appartengono al linguaggio perché sono le codifiche binarie di 3, 6 e 9, mentre 10, 111 e 1011 non appartengono al linguaggio perché sono le codifiche binarie di 2, 7 e 11. La stringa vuota non codifica nessun numero.



Esercizi Espressioni regolari/ER a NFA/Da NFA ad ER (minimizzazione degli stati)

Secondo set di esercizi di Automata Tutor

1) Set di 0/1 alternati {alfabeto 0,1}

For the following regular expression:

(01)*|(10)*|1(01)*|0(10)*

Over the alphabet:

{0,1}

Give **1 word** that the regular expression recognizes and **1 word** that the regular expression doesn't recognize!

words in the regular expression:

- 01010101010

words NOT in the regular expression:

- 0101001011101010 10

Ragionamento:
Chiede 0/1 alternati e si nota che la terza e quarta espressione *unite* (segno di unione per Automata è |)
sono sinonime della prima, da cui il risultato.

2) Tutte le stringhe che contengono un numero pari di “a” (alfabeto {a,b,c})

Your regex: (a(b|c)*a|b|c)*

Ragionamento:
Posso scegliere “a” concatenato a b oppure c in tutte le combinazioni. A questo punto, posso concatenare un altro a oppure b oppure c, in tutte le combinazioni.

3) Tutte le stringhe che NON contengono la sottostringa 101 (alfabeto {0,1})

Your regex: 0*(1*000*)*1*0*

Ragionamento:
In poche parole, copre tutti i casi. Avendo la stringa 0 come iniziale, allora può essere seguita da un 1 e una serie di 0 (almeno uno se non di più).

Se la stringa è vuota (non è 0), passa al pezzo dopo, dove 1 può essere seguito da 0 oppure da 11. Anche i casi in cui una stringa cominci e finisca per 0 sono coperti dai casi esterni alle parentesi.
La parte tra parentesi ha bisogno di 3 zeri per separare tutti gli 1 (1*) dal fatto di avere tutti 0 (0*), quindi avendo anche 100 (con 1*, 0* e 0). Se fosse vuota, non avendo 1000 tra parentesi, accetterebbe anche 101 e non andrebbe più bene.

4) Tutte le stringhe che rappresentano una codifica binaria di un numero multiplo di 3 (alfabeto 0,1)

Your regex: (0|1(01*0)*1)*

Ragionamento:
Dà 8/10, perché dice che non dovrebbe accettare anche la stringa vuota ma dopo numerose prove è l'unica soluzione che ho trovato. Si parta dall'automa, secondo me chiarisce piuttosto che provare alla cieca.
Si nota quindi l'unione iniziale di 0 ed 1. A questo punto si passa con un altro zero e successivamente con ciclo di 1 (1*) alla fine, avendo un altro 0 che torna indietro. Tutto questo è ripetuto più volte. Oltre a questo abbiamo un altro 1 che riconduce all'inizio, che è stato finale.
Da cui tutta sta roba.

5) Tutte le stringhe che contengono 4k + 1 occorrenze di "b" per "k" >= 0
Soluzione:
((a|c)*b(a|c)*b(a|c)*b(a|c)*b(a|c)*)* (a|c)*b(a|c)*
Ragionamento:
Provandole tutte, perché deve avere al minimo un solo b e al più almeno 1+4, la ripetizione di Kleene degli altri simboli è sensata tenerla ripetuta come si vede qui, quindi una per le 4 che sono ripetute di volta in volta. A quel punto, fuori vi sarà la b in più e due volte le espressioni a|c; per qualche ragione, il prof vuole entrambe le possibilità.

6) Tutte le stringhe w che contengono la sottostringa 101 (alfabeto 0,1)

Your regex: (0|1)*101(0|1)*

Ragionamento:
Questo è il più easy; basta proprio avere 101 e poi qualsiasi cifra prima e dopo e va bene.

7) Tutte le stringhe la cui lunghezza è multiplo di 3 (alfabeto a,b,c)

Your regex: ((a|b|c)(a|b|c)(a|b|c))*

Ragionamento:
Qui pure è abbastanza easy, di fatto il multiplo di 3 presuppone che tutto appaia tre volte, in termini appunto di a,b,c. Questo conduce alla soluzione.

Solve Regular expression to epsilon-NFA problem

For the following regular expression:

(ϵ|0)(10)*(1|ϵ)

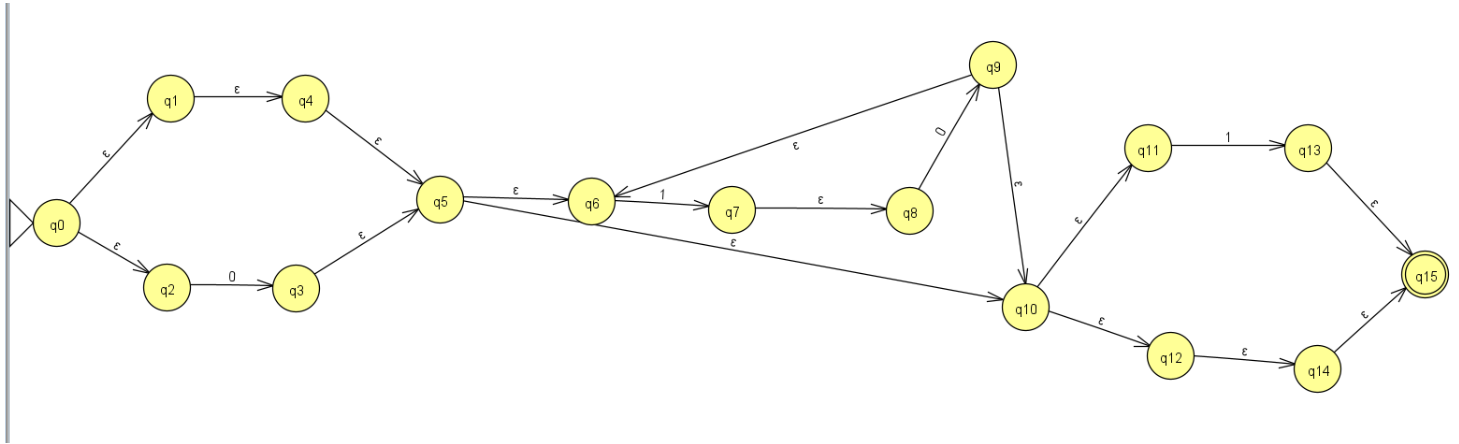
Over the alphabet:

{0,1}

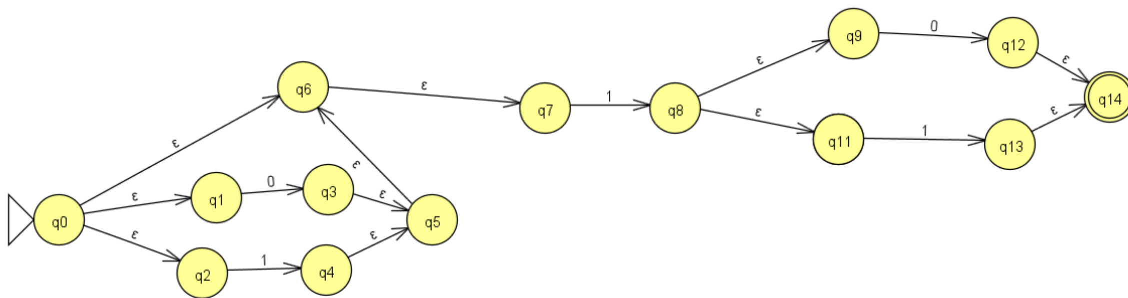
Give an epsilon-NFA that recognizes the same language.

In questo tipo di esercizi bisogna fare le cose a singoli pezzi:
si parte dall'unione ε|0 dopodiché attenzione alla concatenazione 10 e poi lo *.
Bisogna mettere una ε prima di 1, una ε dopo 1, si mette 0, dopodiché lo stato dopo va ad ε, quello stato torna ad 1 per fare lo * e lo stato prima di 1 (ε) va direttamente allo stato dopo lo 0.
Segue poi l'unione 1| ε realizzata come prima.

Parole buttate così ma la sostanza è: scomporre in stati e sapere esattamente come sono fatti gli automi di unione/concatenazione e star.
L'automa che lo rappresenta è:



(0 + 1)∗1(0 + 1) in ε-NFA (da slide)



Scrivere un'espressione regolare per rappresentare il linguaggio sull'alfabeto {a, b, c} che contiene:

9) tutte le stringhe che iniziano con a e sono composte solo di a oppure b;
10) la stringa c

1) a(a+b)*+c
2) a*(a+b)*+c

Altre espressioni regolari varie:

ER che mostra che tutte le stringhe contengano ciascun simbolo almeno una volta

$(a+b+c)^*a(a+b+c)^*b(a+b+c)^*c$

ER per stringhe binarie che contengono almeno tre 1:
(0+1)*1(0+1)*1(0+1)*1(0+1)*

ER per stringhe di testo che descriva le date in formato GG/MM/AAAA
0(1+2…+9)+(1+2)/(1+2+…9)+3(0+1)/(0+1)(1+2)/(0+1+…9)(0+1+…9)(0+1+…9)(0+1+…9)

The set of string over {0,1…9} such that the final digit has appeared before

Let $E = 0+1+\dots+9$

$E^*0E^*0 + E^*1E^*1 + \dots + E^*9E^*9$

The set of string over {0,1…9} such that the final digit has not appeared before:

Let $E_0 = 1+2+\dots+9$

$E_i = 0+\dots+(i-1)+(i+1)+\dots+9 \quad (1 \le i \le 8)$

$E_9 = 0+1+\dots+8$

$E = 0+1+\dots+9$

$E + E_0^+0 + E_1^+1 + \dots + E_9^+9$

Definire un'espressione regolare che descriva l'insieme delle stringhe su $\{a, b, c\}$ contenenti 2 caratteri a o 3 caratteri b.

$(b + c)^*a(b + c)^*a(b + c)^* + (a + c)^*b(a + c)^*b(a + c)^*b(a + c)^*$

Definire un'espressione regolare che descriva l'insieme delle stringhe su $\{a, b, c\}$ contenenti un numero di caratteri c pari a $3k$, per qualche $k \geq 0$.

$((a + b)^*c(a + b)^*c(a + b)^*c(a + b)^*)^*$

Definire un'espressione regolare che descriva l'insieme delle stringhe su $\{0, \dots, 9\}$ che rappresentano interi divisibili per 5

$(0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)^*(0 + 5)$

Definire un'espressione regolare che descriva l'insieme delle stringhe su $\{0, 1\}$ che inziano e terminano con due caratteri diversi.

$0(0 + 1)^*1 + 1(0 + 1)^*0$

ER che accetta tutte le stringhe su {a,b} che contengono esattamente 2 oppure 3 lettere "b"
a*ba*ba*b(a*b+a*)

ER che accetta stringhe con numero di 0 multiplo di 5.
(1+(01*01*01*01*0))*

ER per tutte stringhe binarie che cominciano e finiscono per 1
1(0+1)*1

ER per le stringhe binarie che contengono almeno tre 1 consecutivi
(0+1)*111(0+1)*

ER per le stringhe binarie che contengono almeno tre 1 (anche non consecutivi)

Scritto da Gabriel

(0+1)*1(0+1)*1(0+1)*1(0+1)*
oppure
(0+1)*10*10*1(0+1)*

ER per le stringhe di lunghezza dispari
(0+1)((0+1)(0+1))*

Find a regular expression which represents the set of strings over {a, b} which contains the two substrings aa and bb.
(a+b)*((aa(a+b)*bb)+(bb(a+b)*aa))(a+b)*

Find a regular expression over the language L over the alphabet Σ = {a, b} consisting of strings where the number of b's can be evenly divided by 3.
(a*ba*ba*ba*)a*

Describe which languages the following regular expressions represents, using common english.

(i) $(0 \cup 1)^*01$

(ii) 1^*01^*

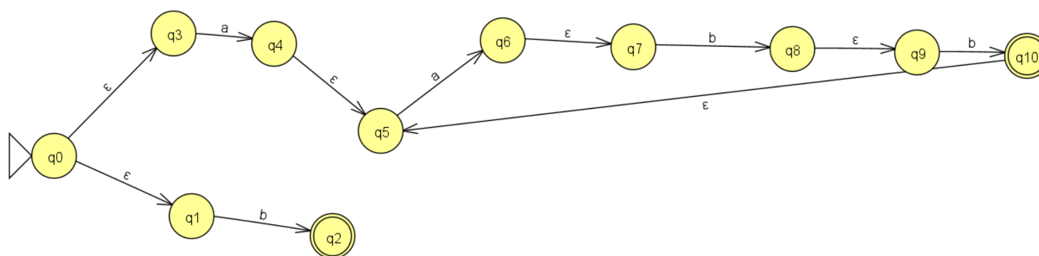
(iii) $(11)^*$

(iv) $(0^*10^*10^*)^*$

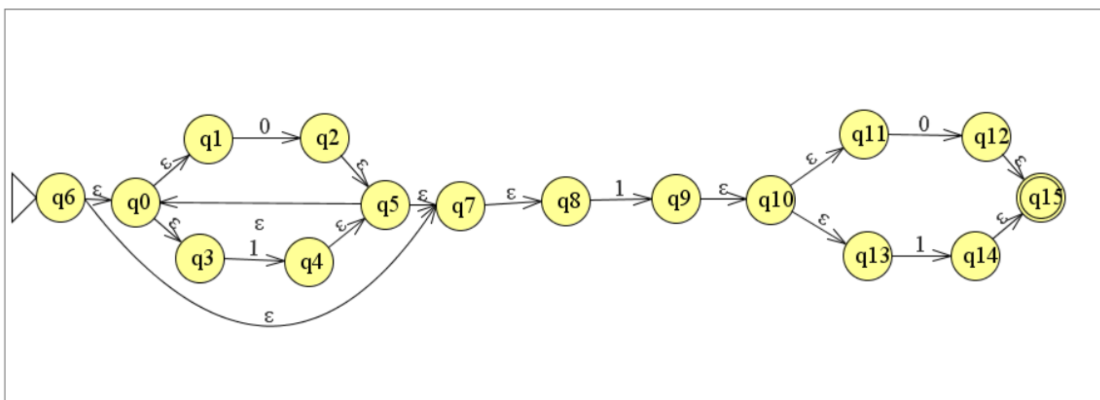
(v) $(0 \cup 1)^*01(0 \cup 1)^*$

(i) Set of strings over {0,1} containing the substring 01
(ii) Set of strings over {0,1} containing at least one 0 and at least two 1s
(iii) Strings consisting only of ones and which lengths are even
(iv) Set of strings over {0,1} containing an even number of 1 and an odd number of 0s
(v) Set of strings over {0,1} containing all the strings and the substring 01

Conversioni di RE in NFA extra:
a(abb)*+ b

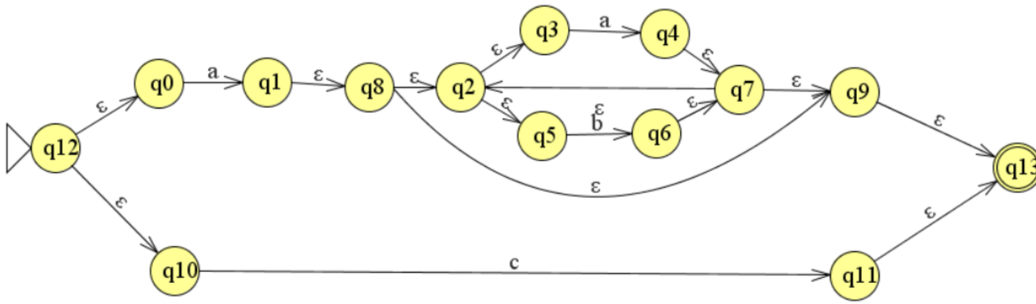


Trasformare (0 + 1)*1(0 + 1) in ε-NFA

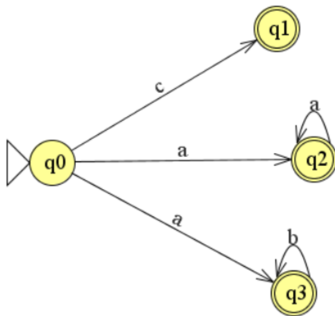


Scritto da Gabriel

a(a+b)*+c

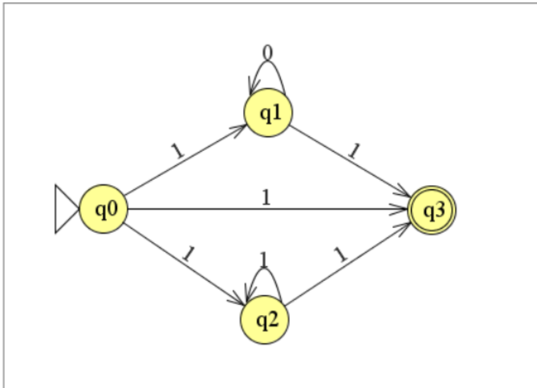


a(a*+b*)+c

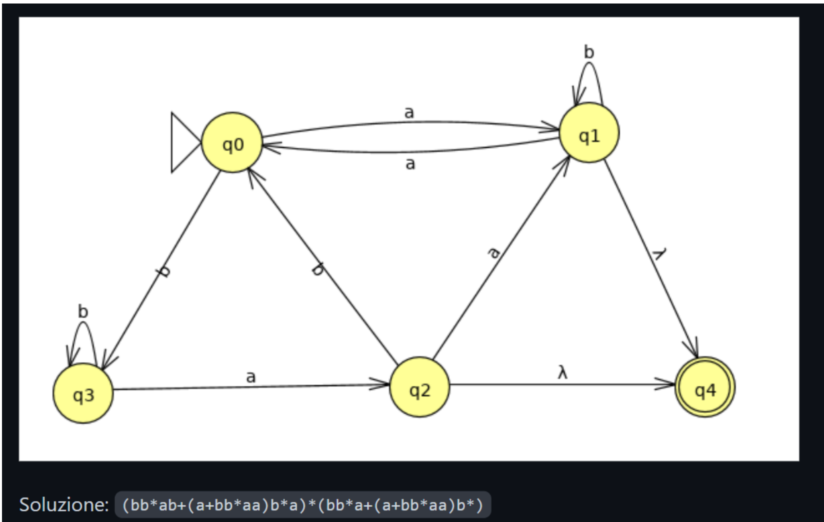


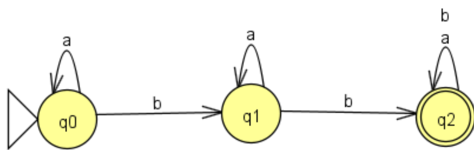
Scrivere una espressione regolare per tutte stringhe binarie che cominciano e finiscono per 1.

1(0+1)*1+1



Conversioni di NFA in RE extra (eliminazione degli stati)

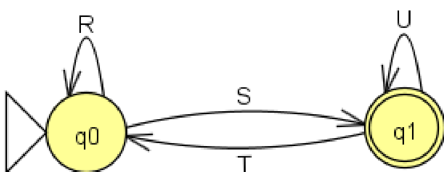




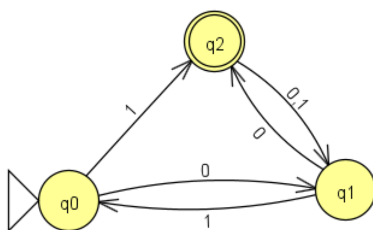
ER risultante: a*ba*b(a+b)*

Passaggi (scritti per non disegnare tutte le volte gli automi e per ricordarmi):
Secondo p: Self loop iniziale con a , freccia seguente con ba*b, self loop con a,b
Terzo p: freccia intermedia con a*ba*b(a+b)*
Poi ER finale.

Nota: la virgola tra a e b viene interpretata come unione, quindi rappresentante a+b



ER risultante: (R+SU*T)*SU*
Passaggi:
Secondo p: con stato iniziale con self loop di R e SU*T, freccia intermedia con S e self loop sul finale con U*
Terzo p: self loop iniziale con R + SU*T, freccia intermedia con S e self loop con U
Poi ER finale.

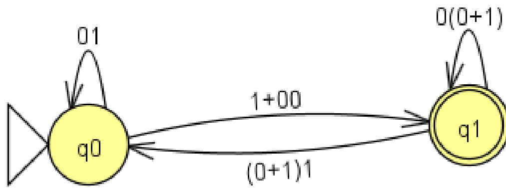


Stati Percorsi

Scritto da Gabriel

q1/q1	Ø+01=01	q1/q2/q1
q1/q0	1+00	q1/q2/q0
q0/q1	Ø+(0+1)1	q0/q2/q1
q0/q0	Ø+0(0+1)0	q0/q2/q1

Fatta così la tabella letteralmente scrive tutto e basta ricopiarla in automa:



Da cui l'espressione regolare è:
(01+(1+00) (0(0+1))* (0+1)1)*(1+00) (0(0+1)*

Tutorato 2

Se avessi come alfabeto {0,1}, 0 da sola è un'espressione regolare.
In questo caso riconosce solo lo 0.
L'ordine di applicazione è:

11) le parentesi
12) unione, Kleene
13) 0, insieme vuoto, stringa vuota

Piccola nota: se avendo un alfabeto mettessi $\sum^*$ sarebbe come scrivere tutto come star.
Esempio: avendo alfabeto 0,1 scrivendo $\sum^*$ sarebbe come scrivere (0+1)*

Esercizio 1

1. Stringhe binarie che iniziano e finiscono con 1
1(0+1)*1 (sarebbe equivalente 1(01)*(11)*(00)*(10)*

2. Stringhe binarie di lunghezza pari
[(0+1)(0+1)]*

3. Stringhe binarie di lunghezza dispari
(0+1)[(0+1)(0+1)]*

4. Stringhe binarie che terminano con 00
(1+0)*00

5. Stringhe binarie in cui il quarto simbolo e' uno zero
(0+1)(0+1)(0+1)0(0+1)

6. Stringhe binarie in cui ciascuna coppia di zeri è seguita da una coppia di uno
(1+01+0011)*(0+ε)

7. Stringhe binarie divisibili per quattro
Si riflette avendo la stringa con almeno 00 alla fine per essere divisibile per 4.
Tipo:
100 10100

L'aggiunta dello 0 permette di avere, intesa come stringa, la rappresentazione anche del caso "ho uno 0 finale" negli elementi dell'insieme.
(0+1)*00+0

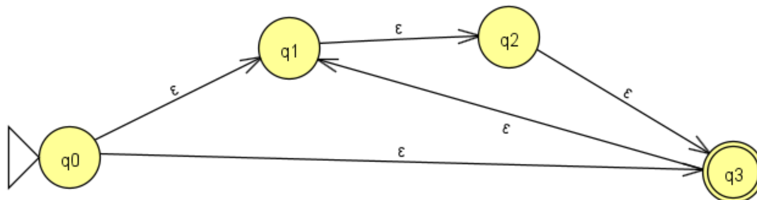
8. Le rappresentazioni di interni binari compresi fra 0 e 4
(0+1)(0+1)

9. Le rappresentazioni di interni binari compresi fra uno e quattro
01+10+11

Esercizio 2

1. L(Ø*) = Ø
FALSO → L'epsilon conta come elemento, nonostante l'insieme vuoto

Per convincerci possiamo fare l'automa:

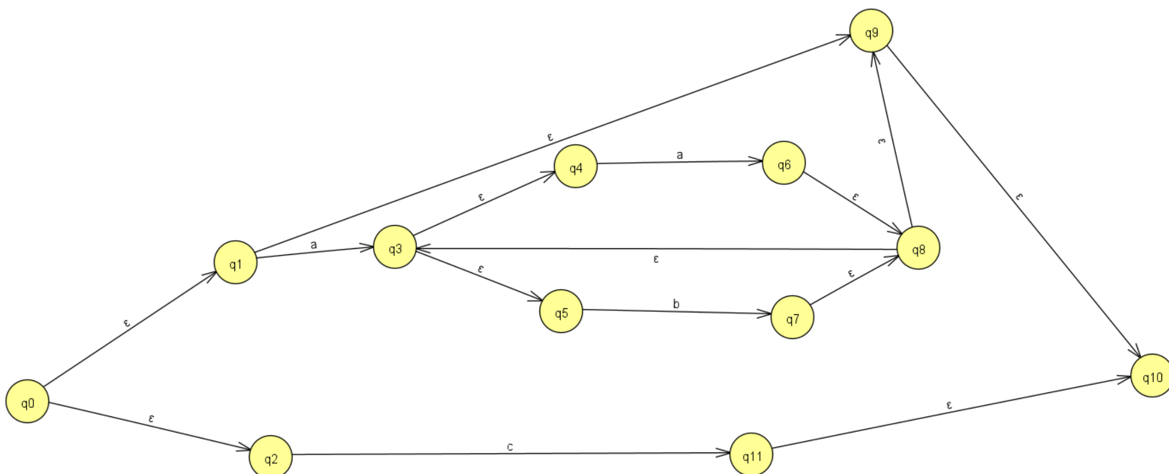


2. La stringa 'baa' appartiene al linguaggio L(a*b*a*b*)
VERO → a* accetta la stringa vuota, poi b* ha "b", a* ha "aa" e b* è vuoto

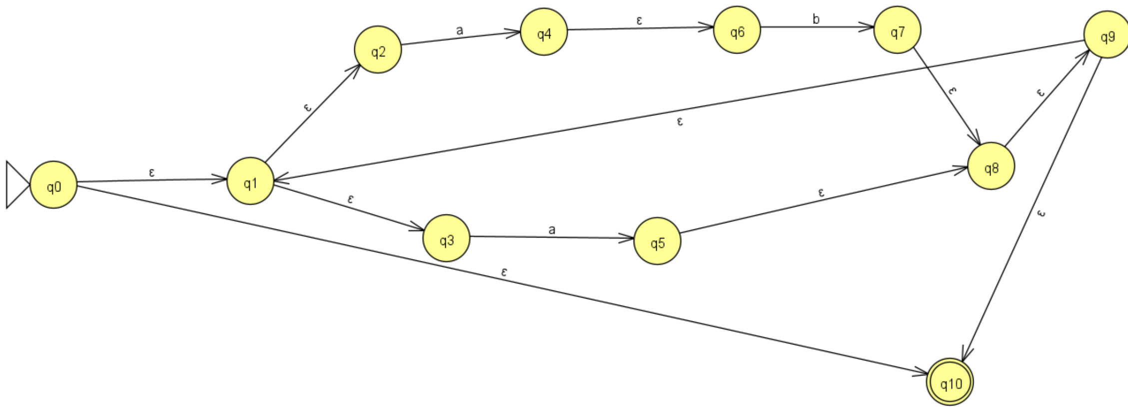
3. L((a+b)*) = L((a*b*)*)
Giulio diceva di dover controllare l'ordine di applicazione delle parentesi. Direi comunque falso, verso l'ordine citato poco sopra. Viene applicato lo stare della parentesi, poi vengono applicate le singole stare sulla seconda espressione, risultando diverso dall'idea originale.

Conversione da RE a FA

1. a(a* + b*) + c



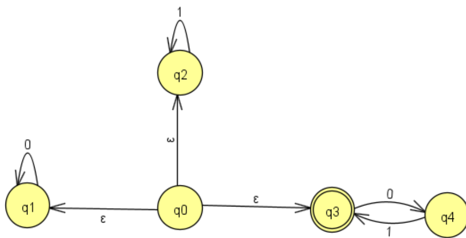
2. (ab + a)*



Costruire un FA che accetti il linguaggio denotato dalle seguenti RE:

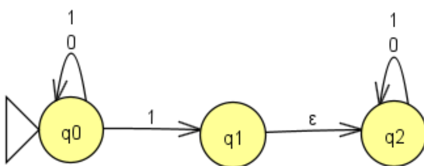
1. $R = 0^* + 1^* + (01)^*$
2. $R = (0+1)^*1(0+1)$
3. $R = a(a+b)^* + c$

1)

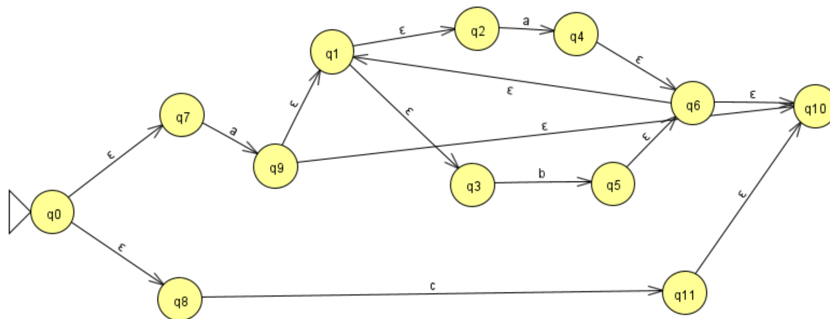


Si segnala q0 come stato iniziale qui in 1) e q2 come stato finale qui in 2)

2)



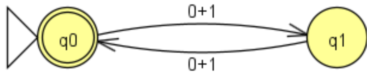
3)



<u>*Conversione da FA a RE*</u>

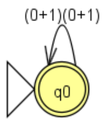
Fornire un FA per i seguenti linguaggi e convertire l'automa in una espressione regolare:

1. Stringhe binarie di lunghezza pari
∑={0,1}



Passaggi:

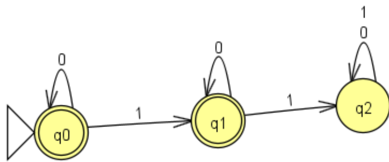
q1 q0 -> q0 (0+1)(0+1)



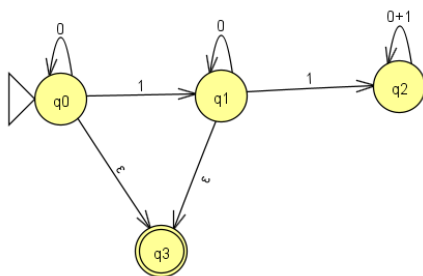
Quindi per renderla corretta: [(0+1)(0+1)]*

2) Automa che riconosce le stringhe con al più il carattere 1 ripetuto una volta sola:

1)

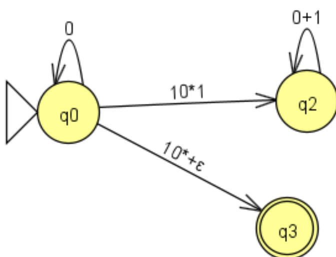


2) RE

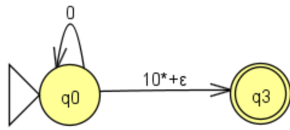


Passaggi:

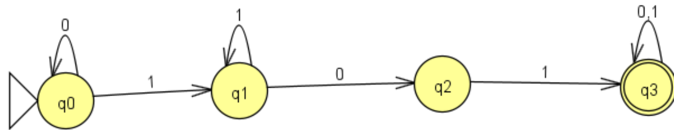
q1	q0->q2	10*1
	q0->q3	10*ε + ε



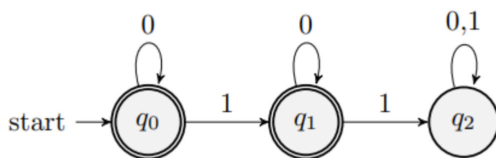
q2 rappresenta uno stato non accettante e possono semplicemente ignorarlo nell'eliminazione. Quindi:

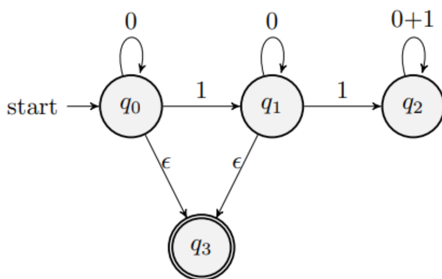


2. Stringhe binarie che non comprendono la stringa 101
L'idea è di definire un DFA che accetta la stringa 101, lo invertiamo quindi facciamo il complemento, quindi:
Caso del "contiene 101":

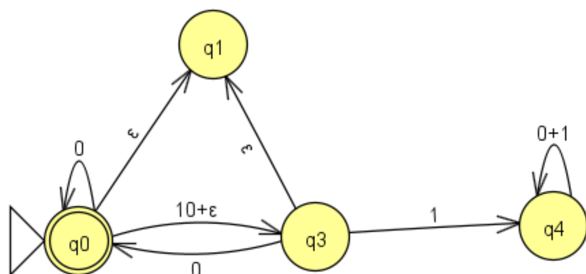


Rispettando le regole, sta formando il complemento (mettendo tutti gli stati non finali come finali) e poi regolarizzando l'automa, facendo in modo la computazione finisca in uno stato accettante, avendo un solo stato iniziale e finale. L'operazione di complemento è chiusa e fattibile quindi. Poi andiamo ad eliminare incrementalmente i vari stati, notando che q2 viene eliminato "senza colpo ferire" essendo uno stato cestino, infatti non è stato accettante ed ogni combinazione di caratteri ottenuta su di esso non sarebbe utile. Si prosegue quindi:

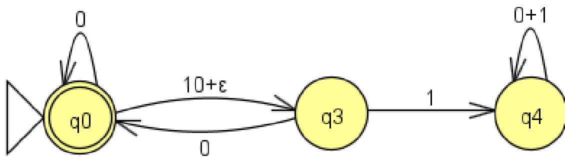




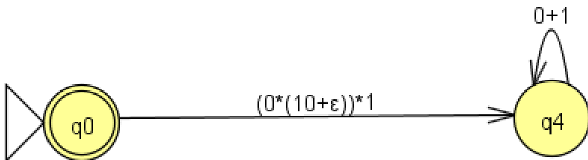
Passaggi:
Elimino q2 q0-q3 10+ε



Elimino q1: q0-q3-q4 (10+ε)



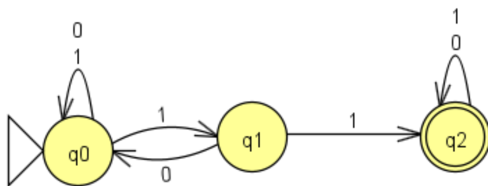
Elimino q3: q0-q4 (0*(10+ε))*1



Elimino q4: (0*(10+ε))*1+(0+1)*

ER: 0*(ε+(10+ε))1(0+1)*

3. Stringhe binarie in cui 0 è seguito da 11 (esercizio originalmente presente e da me lasciato):



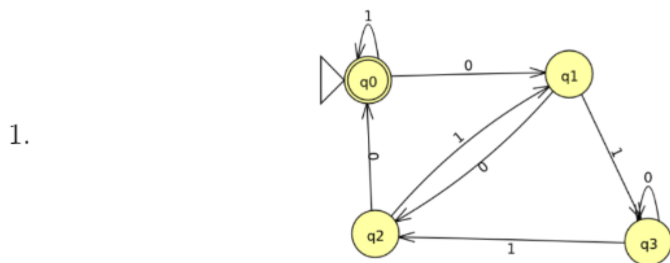
Passaggi (eliminazione degli stati):

Elimino q1	Prec	Succ	
	q0	q2	10(01)* e 1(01)*

Abbiamo inoltre il selfloop su (01)* e concatenando il tutto diventa:
(0+1)*(01(01)*)*1(01)*

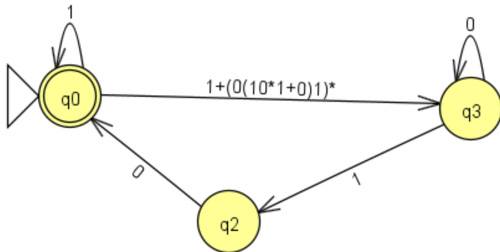
Converti l'automa dato in RE:

1.

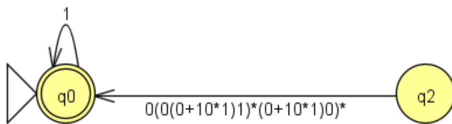


Passaggi:

Tolgo q1 e:
q0-q3-q2-q0 1+(0(10*1+0)1)*
q2-q0



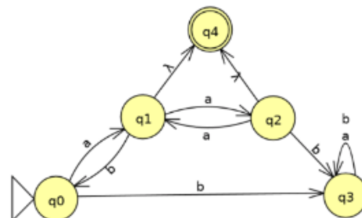
Tolgo q3 e avrò similmente i due percorsi (0)* ed 1, trattati come unione:



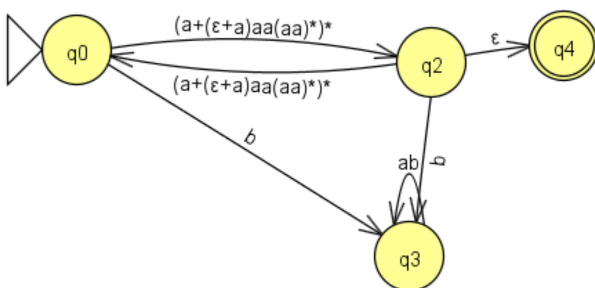
Tolgo infine q2 che non disegno ed otterrò dunque:

1. $R = (1 + 0((0 + 10^*1)1)^*(0 + 10^*1)0)^*$

2.



Eliminiamo q1 e avremo da considerare l'unione di "a" con la "epsilon" e aa (poi ripetuto per concatenazione).
Mettendo tutto insieme andremo a comporre: (a+(ε+a)aa(aa)*)*

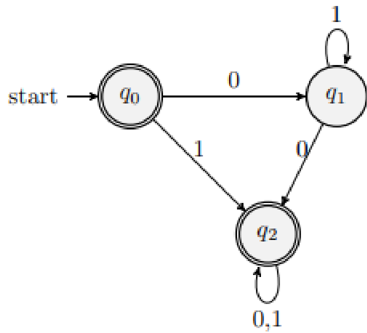


Eliminando q3 perché stato non finale, elimineremo anche q2 e avremo una situazione simile a prima, concatenando in questo caso ((ε+aa(aa)*)ab)
ER finale infatti:

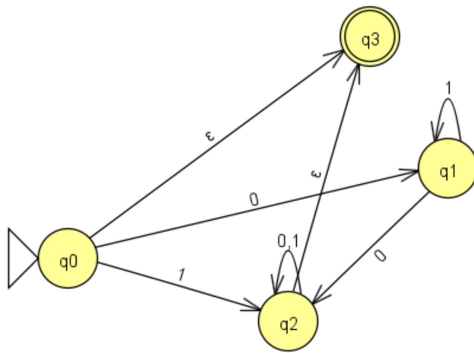
2. $R = ((\epsilon + aa(aa)^*)ab) * (a + aa(aa)^*(\epsilon + a))$

3.2 Ex2

Convertire in RE il seguente automa

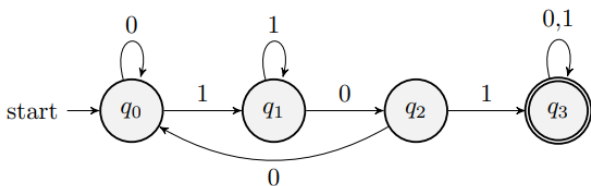


Quello presentato da Giulio è un altro automa! Le regole dicono di avere un solo stato finale ed un solo stato iniziale, aggiungendo una ε-transizione verso il nuovo stato iniziale e/o finale.
In questo caso si ha un solo stato finale convertendo tutti gli stati non finali., così:



A questo punto procediamo come sempre, provando ad esempio ad eliminare q1 e successivamente q2
Con q1 il percorso è: 01*0
Con q2 il percorso è: 1 (self-loop 0,1 su q2 ripetibile, quindi (0+1)*.
Diventerà quindi l'espressione 1+01*0(0+1)*

Il risultato è identico a quello di Giulio, leggermente diverso:



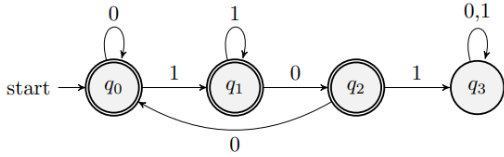
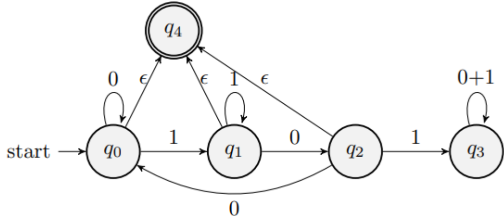
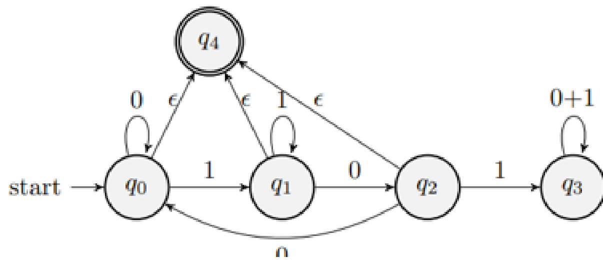
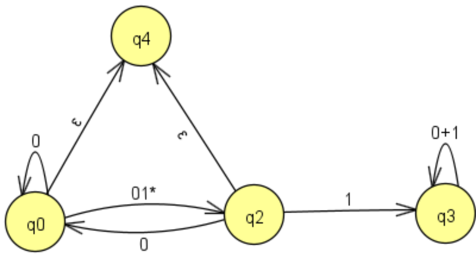


Figure 9: DFA inverso per lingugaggio di partenza





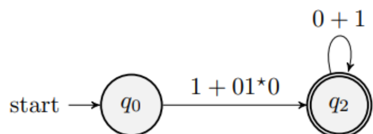
A quest'ultimo automa invece, andiamo a fare le eliminazioni di stati.
Eliminiamo q1 e si vede che esso presente 1 come stato entrante, 1 che va verso sé stesso e 0 in uscita.
Quindi andremo a mettere 01*



A questo punto si nota chiaramente che abbiamo 01*0 concatenato al precedente.
Se eliminiamo q2 abbiamo:

14) la parte che esce con 0 (01*0)ù
15) la parte che esce con 1 (0+1)*, unita alla precedente perché q2 ha più stati uscenti

Quindi avremo:



Da cui poi si ha l'espressione: 1+01*0(0+1)*

3.3 Esercizi aggiuntivi

3.3.1 Ex1

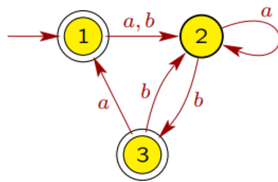
Scrivere l'espressione regolare per i seguenti linguaggi.

1. tutte le rappresentazione di interni binari compresi fra zero e otto
2. tutte le rappresentazione di interni binari compresi fra uno e otto

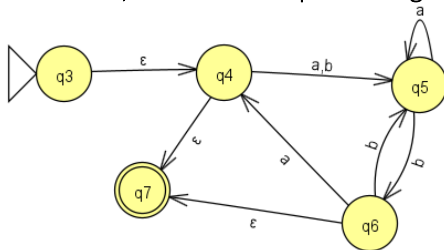
1) 0000 + 0001 + 0010 + 0011 + 0100 + 0101 + 0110 + 0111 + 1000
2) 0001 + 0010 + 0011 + 0100 + 0101 + 0110 + 0111 + 1000

<u>Altri esercizi vari:</u>

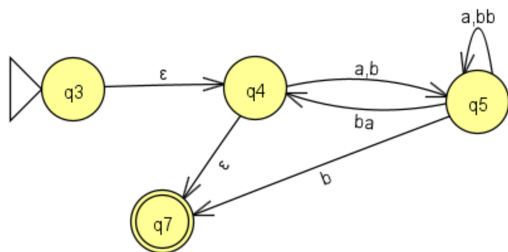
2. Use the procedure described in Lemma 1.60 to convert the following DFA M to a regular expression.



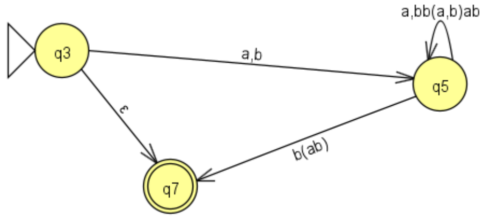
Come al solito, andiamo ad operare togliendo il doppio stato finale (questo si chiama GNFA):



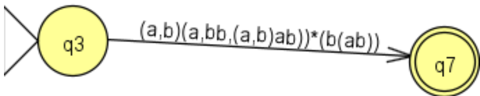
Eliminiamo per esempio q6
q5 bb
q4 ba

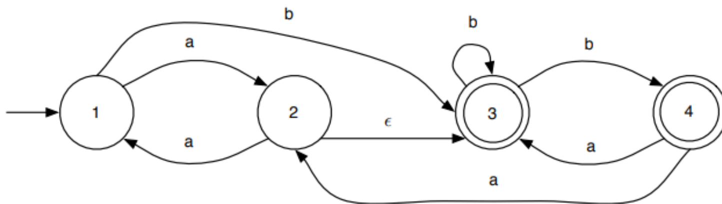


Cerchiamo quindi di eliminare q4
q3-q4-q5-q7 ε(a,b)
q3-q4-q7 ε
q5-q4-q5 (a,b)ab
q5-q4-q7 ab

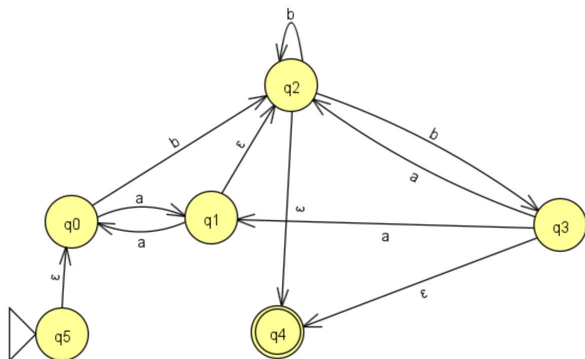


Andremo quindi ad eliminare q5
Quindi avremo (a,b)(a,bb,(a,b)ab))*(b(ab))





Riduciamo di nuovo mettendo un solo stato finale (in pratica si toglie 3 idealmente come stato finale e poi ne si aggiunge uno nuovo a cui i vecchi stati accettanti vanno con delle ε-transizioni.
Oltre a ciò, visto che lo stato iniziale ha transizioni entranti, si aggiunge un nuovo stato iniziale buttandogli poi la ε-transizione.

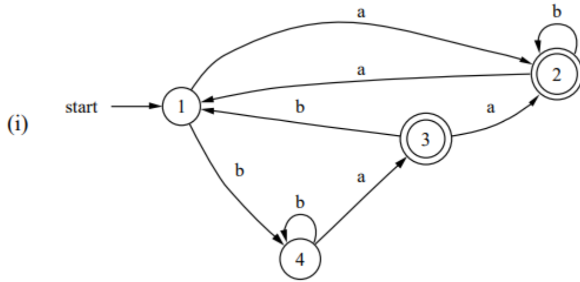


Voglio quindi eliminare q0.
q1 aa
q2 b
Si ha inoltre una sorta di ciclo tra q0,q1,q2, quindi ab+ε

Exercise 3.12. (o27)

Construct regular expressions which are equivalent to the following (deterministic) finite automata:

(i)



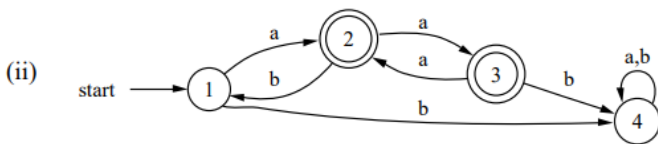
Una volta fissato un unico stato finale 5, andando con ε verso di esso,
ho proceduto eliminando:

4 – $b^+a(b+ab^*a))^*$ (due percorsi entranti ed uscenti)
2 – $ab^*+(\varepsilon+ab^*)$ (due percorsi entranti)
3 – $b+a$ (unito a prima sempre con il ciclo su sé stesso e poi con la transizione uscente)

Ottenendo infine:

(i) $\left(b^+a(b \cup ab^*a)\right)^*\left(ab^* \cup b^+a(\varepsilon \cup ab^*)\right)$

(ii)

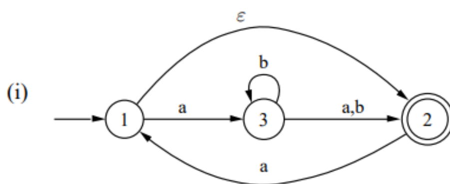


Considerando lo stato 4 come stato non accettante, si fa solo attenzione al self-loop a,b
utile per quello che stiamo facendo.

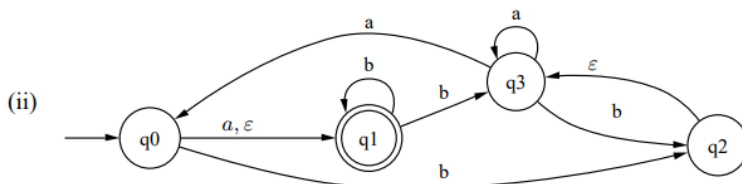
Exercise 3.14.

Convert each of the NFA's below to an equivalent DFA by subset construction.

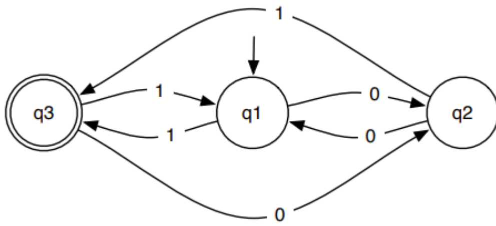
(i)



(ii)

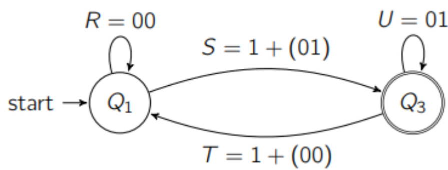


Design a RE accepting the same language as:



Absorbing transitions	$(Q, Q\prime) : E$	AX^*B	Result
$(\mathbf{Q_1}, \mathbf{Q_1}) : \mathbf{R}$	$\emptyset$	00	00
$(\mathbf{Q_1}, \mathbf{Q_3}) : \mathbf{S}$	1	01	$1 + (01)$
$(\mathbf{Q_3}, \mathbf{Q_3}) : \mathbf{U}$	$\emptyset$	01	01
$(\mathbf{Q_3}, \mathbf{Q_1}) : \mathbf{T}$	1	00	$1 + (00)$

Give the automaton:

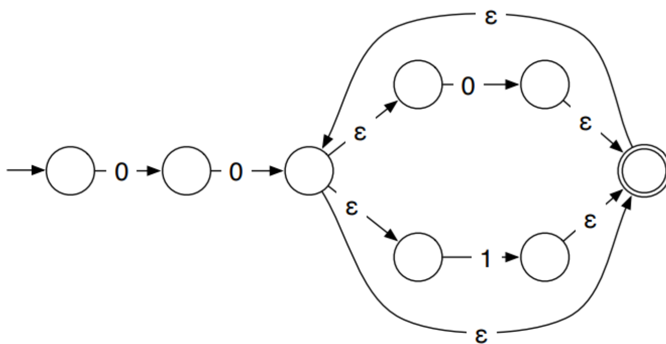


by applying the rule: $(R + SU^*T)^*SU^*$, the RE is:

$$((00) + (1 + (01))(01)^*(1 + (00)))^*(1 + (01))(01)^*$$

Conversione di ER in Automa:

3. $00(0 + 1)^*$



Pumping lemma

Dal file "gioco.pdf" (adattando il file a richieste esplicite se siano espressioni regolari o meno) e dalla slide *"04-pumpinglemma.pdf"*

L = {w ∈{a, b}∗ : il numero di a è uguale al numero di b}
Il linguaggio è regolare?

Supponiamo per assurdo sia regolare. Dunque, data k la lunghezza pumping, consideriamo la parola $a^k b^k$, su cui scegliamo una suddivisione di w tale che:
y ≠ ε e |xy| ≤ k:
Prendiamo ad esempio:

w=	aaaa	a…..a	a..aa..b…bb
	x	y	z

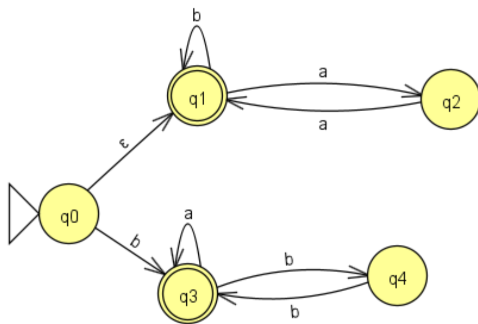
Si sa infatti che $xy^iz \in L(A)$, ma possiamo notare che il numero di a diventa più grande di quello di b per esempio con $xy^2z \in L_{ab}$ e quindi non è più verificata la seconda delle due condizioni dette prima.
Morale della favola: non è regolare.

L = {w ∈{a, b}∗ : il numero di a è maggiore del numero di b}
Il linguaggio è regolare?

Ripetendo lo stesso discorso di prima, valenti le tre condizioni, supponiamo l'espressione lo sia e consideriamo come parola:
$w=a^{(k+1)}b^k$
Prendendo come prima una suddivisione generica della parola w,
tipo xy^0z, avremo che il numero di a è invece inferiore a quello di b.
Morale della favola: non è regolare.

L = {w ∈{a, b}∗: numero di a è pari, il numero di b è dispari}
Il linguaggio è regolare?

Supponiamo per assurdo non sia regolare e consideriamo come parola:
$w= a^{(2k)}b^{2k+1}$
Si nota però che è possibile costruire un automa equivalente del tipo:



Inoltre, si può rappresentare regolarmente con l'espressione:
(a(a+b)*a(a+b)*b(a+b)*)*
Morale della favola: l'espressione è regolare

L_{2ab} = {w ∈{a, b}∗ : numero di a è due volte il numero di b}
Il linguaggio è regolare?

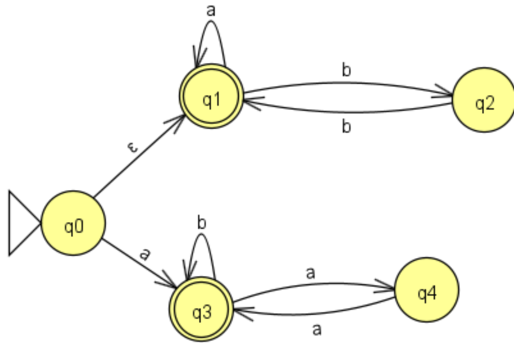
Supponiamo per assurdo che lo sia e, data una generica *h* lunghezza del PL,
consideriamo la parola $w = a^{2k}b^k$
Considerando poi un generico split:

w=	aa…aa	a..a	aa…. b
	x	y	z

Si nota che il numero di a è maggiore rispetto a quello di b e le stringhe x ed y sono fatte di sole "a"
Morale della favola: non è regolare

L = {w ∈{a, b}∗: numero di a è dispari, il numero di b è pari}
Il linguaggio è regolare?

Supponiamo per assurdo che lo sia e consideriamo come parola:
$w= a^{(2k+1)}b^{2k}$
Il caso è paritetico a quello precedente, ma con i termini girati e:



L'ER che può rappresentarlo è:
(a(a+b)*b(a+b)*b(a+b)*)*
Morale della favola: l'espressione è regolare

Slide:

- $L_n = \{1^{3n+2} : n >= 0\}$ è regolare?

Consideriamo una pumping length *k* e consideriamo la parola $w = 1^{3k+2}$, tale che $x \neq \varepsilon$ e $|xy| <= k$.
Poniamo una suddivisione, per due esponenti q,p > 0 tali che:
$x = 1^q$ $y=1^p$ $z=1^{3k+2}$
Eseguendo un pumping su xy^0z avremo:
$w=1^{3k+2-q-p}$ che appartiene al linguaggio.

Qualora decidessimo di attuare un pumping, si vedrebbe che il numero di stringhe è sbilanciato.
Con pumping i=3:
$w=1^{3k+2-3p-q}$ che non appartiene al linguaggio, in quanto, per una scelta diversa del numero di 1 da x e da y non sarebbe nel linguaggio.
Tuttavia si rappresenta con un DFA e il linguaggio è regolare.
Se fallisse per tutte le decomposizioni, allora il linguaggio sarebbe non regolare.

- $L_{mn} = \{0^n1^m0^n : m + n > 0\}$ è regolare?

Consideriamo una pumping length *k* tale che possiamo immaginare $w=0^k1^p0^k$ per un k > 0.
A queste condizioni, applichiamo un pumping xy^0z in maniera tale da avere $0^{(p-k)}10^p$.
In tali condizioni, la somma m + n sarebbe sbilanciata nel numero di 0 e chiaramente non appartiene al linguaggio, rendendolo non regolare.

- $L_{mnp} = \{0^n1^m0^p : m + n + p > 0\}$ è regolare?

Consideriamo una pumping length *k* tale che possiamo immaginare $w=0^k1^p0^q$ per un k, p, q> 0.
A queste condizioni, si vede che gli 0 saranno tutti in *x* mentre la condizioni |xy| <= p implica che xy consistono solamente di 0.
Eseguendo un generico pumping su xy^iz, tale da avere $0^k1^{p+i}0^q$ avendo quindi che j + k + i < k. Ciò no accade per contraddizione, dato che i >= 1 e le stringhe sarebbero chiaramente sbilanciate. Dunque il linguaggio non è regolare.

- $L_{2ab} = \{ w \in \{a, b\}^* :$ numero di a è due volte il numero di b$\}$ è regolare?

Assumiamo per assurdo L sia regolare e abbiamo k la pumping length. Scegliamo $w=a^{2k}b^k$.
Se w rispetta le condizioni |xy| <= k e ed y non vuoto, allora deve esistere almeno una "a" dentro al linguaggio. Tuttavia eseguendo un pumping qualsiasi, come ad esempio, xy^3z, avremmo che il numero delle a è decisamente spareggiato rispetto alle b, almeno il doppio se non di più, non rispettando quanto richiesto. Dunque il linguaggio non può essere regolare.

Varie:

1. $L = \{0^m 1^n, m \leq n, n, m \in \mathbb{N}\}$
2. $L = \{0^m 1^n, n \neq m, n, m \in \mathbb{N}\}$
3. $L = \{x \in \{a, b\}^* |$ il numero di a è minore del numero di b $\}$

1) Supponiamo per assurdo sia regolare e data h la lunghezza pumping, ipotizziamo come parola, sapendo inoltre m<=n:
   $w=0^2 1^3$
   inoltre avendo come possibile suddivisione:
   0 0 111
   x y z
   Si nota già come il numero di 1 sia maggiore di quello degli 0
   Dunque questa espressione non è regolare.

2) Supponiamo per assurdo sia regolare e data h la lunghezza pumping, ipotizziamo come parola, sapendo inoltre n diverso da m:
   $w=0^h 1^{h+m}$
   Si può notare che scegliendo una qualsiasi suddivisione xyz con le solite due condizioni, essendo che l'esponente di 0 è diverso da quello di 1, si avrà per forza uno sbilanciamento,
   avendo quindi più 0 oppure più 1.
   Dunque anche questa non risulta essere regolare.

3) Supponiamo per assurdo sia regolare e data h la lunghezza pumping, ipotizziamo come parola,
   $w=a^h b^i$ con h < i
   Con una generica suddivisione xyz tale che y diverso da vuoto e xy <= k avremo che:
   w= a..aa a..b b…b
   x y z
   Poiché xy <= k, le stringhe x ed y hanno più b rispetto alle a
   Quindi anche qui si ha una espressione non regolare.

I seguenti linguaggi sono regolari? Motivare la risposta.

1. $L = \{0^{5k}, k \in \mathbb{N}\}$
2. $L = \{0^{pq}, p, q \in \mathbb{N}, p > 1, q > 1\}$
3. $L = \{0^{k^3}, k \in \mathbb{N}\}$

1) Il linguaggio risulta essere regolare; se noi per ipotesi considerassimo k come 1 avremmo:
0^5
Se la considerassimo come espressione sotto potenza, dunque si tratterebbe di una semplice:
(00000)*, rappresentabile proprio con la precedente ER e quindi *l'espressione risulta a sua volta regolare*

2) Supponiamo per assurdo sia regolare e, avendo come al solito una parola $w=0^2 1^3$, quindi due esponenti generici, ci aspettiamo che nella parola w, scegliendo una generica suddivisione xyz, avremo una parola che non avrà la solita condizione |xy| <= k come rispettata. Pertanto *l'espressione risulta non regolare*.

3) Supponiamo per assurdo l'espressione sia regolare, quindi supponendo h solita lunghezza pumping e la parola w, se prendiamo il caso base 0^{h^3} viene soddisfatta, perché il numero di 0 è sempre minore di k.
Se avessimo quindi la classica suddivisione xyz, con |xy| <= h, supponiamo che sia tutto composto di soli 0 e avremo per esempio, una suddivisione
$x=0^n$ $y=0^m$ $z=0^{h-n-m}$
Possiamo come sempre pompare y, quindi ad esempio $w'=xy^3z$, quindi ad esempio $0^n 0^{3m} 0^{h-n-m}$

avendo poi $w=0^{2m+h}$. Di fatto questa roba serve a dire che 0^{h^3} è maggiorato a sua volta da una quantità che *non lo rende regolare*.

I seguenti linguaggi sono regolari? Motivare la risposta.

1. $L = \{a^n | n = 2^k, k \in \mathbb{N}\}$
2. $L = \{0^n 1^m 0^{m+n} | n, m \in \mathbb{N}\}$
3. $L = \{0^n 0^{2^n} | n \in \mathbb{N}\}$
4. $L_{pal} = \{x \in \{0,1\}^* |$ x è palindrome $\}$
5. $L = \{x \in \{0,1\}^* |$ il numero di 0 è uguale al numero di 1 $\}$

1) Intuitivamente l'espressione non risulta essere rappresentabile da automi/espressioni. Proviamo a dimostrarlo per il PL, considerando una parola w, tale che w >= h.
Sapendo che n deve essere 2^k, poniamo come parola base proprio $w=a^{2^k}$.
Scegliendo poi xyz, con |xy|<=h, avremo una suddivisione del tipo:
$x=a^n$ $y=a^m$ e z come al solito formata dalla parte rimanente.
L'unica cosa che possiamo fare è trovare una parola pompata su y tale che non sia nel linguaggio.
Per esempio si può provare con la parola a^{2m} quindi con xy^2z; questa parola sta nel linguaggio perché è ancora valida la condizione iniziale (cioè a^n tale che $m=2^k$).
Se invece provassimo con xy^0z, otterremmo una parola senza y e solo con a^n che dovrebbe essere uguale a 2^k. Qua in effetti se appartenga o meno al linguaggio dipende dal valore di n all'interno di x.
A questo punto torna utile l'osservazione di confronto, basata su "x" < h per proprietà, cioè:
$0 <= 2^h - h <= 2^h - m <= 2^{h+1}$
Si nota quindi che m è sovrastato da h e per confronto con esponenti successivi si dimostra asintoticamente che xy non è <= z, per una scelta degli esponenti come descritto.
Quindi il linguaggio non è regolare.

2) Supponiamo per assurdo il linguaggio sia regolare e come al solito, considerando una parola w, tale che w >= h, con la solita h come pumping length: $w=0^p0^q0^{p+q}$
Proviamo a pomparla con i, con $w=xy^0z$, per cui avremo 0^{2p+q}. Di fatto come sempre xy non sarà <= k, perché avremo 2p+q che non è <= ad n+m.
Quindi il linguaggio non è regolare.

3) Supponiamo per assurdo il linguaggio sia regolare e come al solito, considerando una parola w, tale che w >= h, con la solita h come pumping length: $w=0^h0^{2m}=0^{h+2m}$
vedendo che in effetti l'idea è che "n" sia equamente distribuito tra le parti
e prendendo $w=xy^0z$ vediamo che falsifica sempre l'idea di |xy| <= z,
con $w=1^h0^{h-m}$ non avendo più l'equa suddivisione descritta.
Quindi il linguaggio non è regolare.

4) Supponiamo per assurdo il linguaggio sia regolare e come al solito, considerando una parola w, tale che w >= h, con la solita h come pumping length tipo w: $0^h\ 10^h$
Avendo poi w=xyz descritto come:
$x=0^n$ $y=0^m$ con n+m <= h
Avendo poi z che è come sempre la parte rimanente ($0^{h-m}110^h$) si nota che non è equamente suddiviso, sempre per il magico |xy| <= h, avendo come al solito un assurdo.
Quindi il linguaggio non è regolare.

5) Supponiamo per assurdo il linguaggio sia regolare e come al solito, considerando una parola w, tale che w >= h, con la solita h come pumping length tipo w: a^hb^h
Avendo poi w=xyz descritto come:

$x=0^n$ $y=0^m$, pompata con $w=xy^0z$ avremo che il linguaggio non è regolare, perché avremo un numero maggiore di 0 rispetto agli 1 e ciò contrasta come sempre |xy| <= z, generando l'assurdo.
Quindi il linguaggio non è regolare.

(a) $L := \{a^i b^j a^{ij} \mid i, j \geq 0\}$

Evitando di scriverlo ogni volta:
Assumendo una pumping length p > 0, vogliamo una stringa |w| appartenente al linguaggio con condizioni:

- $|xy| \leq p$.
- $y \neq \epsilon$.
- $xy^k z \in L$ for all $k \in \mathbb{N}_0$.

Anche xy^0z deve appartenere al linguaggio, infatti:
$a^{k-j}b^k a^{k\wedge 2}$
usando k perché j=0 considera che si usi per forza l'altro esponente. Come sempre si viola la proprietà di xy<=z pertanto il linguaggio non è regolare.

(b) $L := \{b^2 a^n b^m a^3 \mid m, n \geq 0\}$

Notando la parola diventerebbe facilmente $a^{n+3}b^{2+m}$
In pratica si vede che, già così, potrebbe diventare facilmente espressione del tipo $b^2a^*b^*a^3$
Se invece prendessimo un esempio di applicazione concreto, considerabile induttivamente vero, con m=2, n=3:
b^4a^6 si vede già che il numero di lettere di una parte può essere sbilanciato rispetto all'altra.
Il linguaggio è quindi regolare.

$L := \{a^{k^3} \mid k \geq 0\}$

Supposte le solite condizioni, possiamo pensare ad una suddivisione della parola del tipo:
Ponendo n=k (invece considero p come pumping length):
$a=^{n\wedge 3}$ $y=a^y$ $z=a^{k-p}$
Di solito, piccola nota, in questi casi si ragiona che y ha il suo esponente e z rappresenta sempre la sottrazione di k per lo stesso esponente di y.
Quindi avremmo: $a^{n\wedge 3+y(k-1)}$
Si vede quindi anche qui che il numero di esponenti possibili può oltrepassare k
ed il linguaggio non è regolare.

Find the minimum pumping length of the languages $L(\mathcal{R})$ where

(a) $\mathcal{R} = \mathcal{R}_1 := 0^*101^*$.

La lunghezza minima è 3 per rispettare le condizioni del PL ed eventualmente pompare.
Infatti, potremmo avere vari scenari, avendo la stringa che inizia o con 0 o con 1 e si sceglieranno almeno due caratteri ripetuti (es. 11*0, 00*1, 01*1, ecc.).
Date queste ipotesi, dunque, la stringa è pompabile e sbilanciabile solo partendo da 3 caratteri in poi.

(b) $\mathcal{R} = \mathcal{R}_2 := 10^*1$

La lunghezza minima qui è 3, in quanto potremmo avere come stringa 11 che non è sbilanciabile e rimane sempre regolare. Invece se avessimo almeno 3 caratteri, è possibile cominciare a pompare ed eventualmente ottenere un linguaggio non regolare, sbilanciando le stringhe.

(c) $\mathcal{R} := \mathcal{R}_1 \cup \mathcal{R}_2$

Dati i due precedenti (appena fatti) linguaggi regolari R_1 ed R_2, l'unione formata dai due linguaggi dovrà rispettare le condizioni di PL.
Similmente a prima, l'unione sarà formata dal numero massimo di stringhe di R_1 ed R_2 e, come tale, può considerare il caso in cui uno dei due linguaggi potrebbe essere nullo (quindi si eseguirebbe il pumping su uno dei due, avendo come lunghezza minima 3). Ciò sarebbe considerabile sufficiente in termini di risultato.
Più formalmente, avremmo ad esempio length(L_1) > length(L_U), avendo un k | per ogni parola pompata, sarà maggiore della lunghezza dell'altro.
Dunque la lunghezza minima è 3.

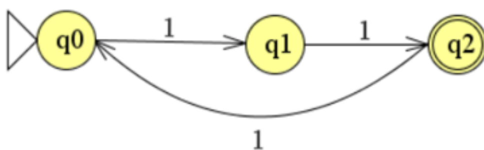
Dal file "04-gioco2.pdf"

Il linguaggio $L = \{1^{3n+2} : n \geq 0\}$ *è regolare?*

- se pensi che il linguaggio **sia regolare**, gira il foglio
- se pensi che il linguaggio **non sia regolare**, riempi lo schema di soluzione qui sotto

Sì, L_{3n+2} è regolare:

- è rappresentato dall'espressione regolare $(111)^*11$
- e riconosciuto dall'automa



Esercizio 6

Il linguaggio $L = \{0^n 1^m 0^n : m + n > 0\}$ *è regolare?*

- se pensi che il linguaggio **sia regolare**, riempi lo schema di soluzione qui sotto
- se pensi che il linguaggio **non sia regolare**, gira il foglio

Sì, il linguaggio è regolare perché è riconosciuto dall'automa a stati finiti

Ripetendo lo stesso discorso di prima, valenti le tre condizioni, supponiamo l'espressione lo sia e consideriamo come parola:
$w=0^{2n}1^{3m}$ (diventa 2n per proprietà alle potenze, somma di n+n)
Prendendo come prima un qualsiasi split w=xyz, y ≠ε, |xy|<=k
Mettendo per esempio i = 3 nella formula xy^iz
avremo che il numero di 0 sarà sempre maggiore di quello di 1 e non è regolare.
Morale della favola: non è regolare

Esercizio 7

Il linguaggio $L = \{0^n 1^m 0^p : m + n + p > 0\}$ *è regolare?*

- se pensi che il linguaggio **sia regolare**, riempi lo schema di soluzione qui sotto
- se pensi che il linguaggio **non sia regolare**, gira il foglio

Supponiamo per assurdo sia regolare e diciamo che, data h la lunghezza del P.L. consideriamo la parola:
$w = 0^{n+p}1^m$
Prendendo come al solito un qualsiasi split w=xyz , y ≠ε, |xy|<=k
Mettendo per esempio i = 3 nella formula xy^iz
Con un qualsiasi i, vedendo che nella xyz ci sarebbe di sicuro una possibile intersezione,
il numero di 0 sarà sempre maggiore di quello degli 1.
Morale della favola: non è regolare

Esercizio 8

Il linguaggio $L = \{w \in \{a, b\}^* :$ *numero di a è due volte il numero di b*$\}$ *è regolare?*

- se pensi che il linguaggio **sia regolare**, riempi lo schema di soluzione qui sotto
- se pensi che il linguaggio **non sia regolare**, gira il foglio

Supponiamo per assurdo sia regolare e diciamo che, data h la lunghezza del P.L. consideriamo la parola:
$w = a^{2n}b^n$
Prendendo come al solito un qualsiasi split w=xyz , y ≠ε, |xy|<=k
Si può osservare che non si rispetta la condizione 3 (|xy|<=k)
in quanto il numero di a, dati gli esponenti di prima, con un qualsiasi i dentro xy^iz
sarebbe maggiore del numero di "b"
Morale della favola: non è regolare

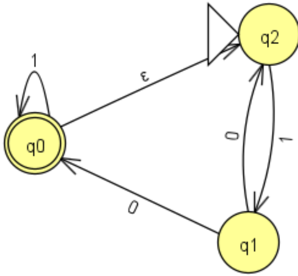
<u>Tutorato 3</u>

1) Linguaggio regolare:

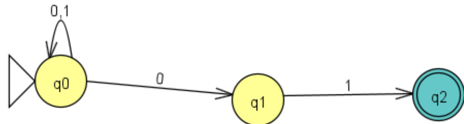
$w = w_1 \ldots w_n$
$w^n = w_n \ldots w_1$

A, $A^R = \{w^r | w \in A\}$
(0+1)*01|10(0+1)*
Di conseguenza esiste almeno un DFA che lo rappresenta tutto.

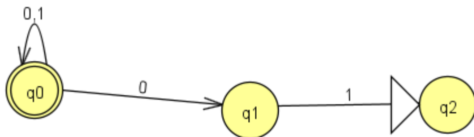
Possiamo quindi invertire la rappresentazione, scambiando stato iniziale con stato finale:



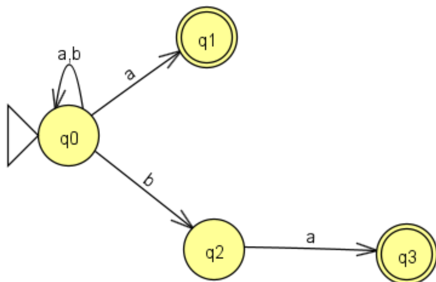
So però che posso convertire un DFA ad un NFA:



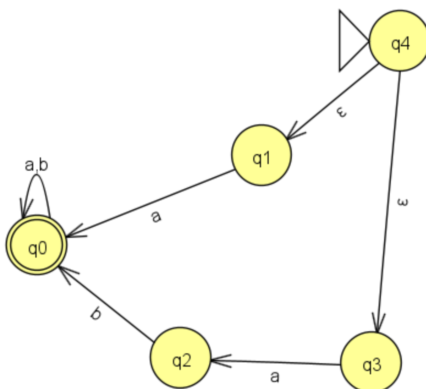
Il corrispondente NFA è:



Possono esistere quindi NFA con più stati accettanti come ultimo stato.



Qui si parte dall'NFA; parto modificando la funzione di transizione e aggiungendo stati.
Il linguaggio è regolare perché lo trasformo in automa e poi aggiungo nuovi stati con transizioni:



Se il linguaggio è regolare, vale il PL.
Se invece vale il PL, non si può dire che il linguaggio regolare.
Esempio: Unione Europea ed euro. Non tutti i paesi hanno l'euro.

Posso però dire che:
A è un LR, quindi esiste un p tale che per ogni w in A, w>=P.
W= xyz
y > 0
xy <= p
∀ c >= 0 $xy^iz \in A$
Si dimostra poi che non è regolare
L'ipotesi: B è regolare
∀ i >= 0 $xy^iz \in A$
Basta questa per pompare e dimostrare non appartiene ad un linguaggio.
Due tipologie di esercizio:

16) togliere un "pezzo" e dimostrare l'esercizio
17) trovare particolare operatori matematici che dimostrino, perché può non valere per tutti i casi

1 Linguaggi regolari

1. Per una stringa w $= w_1 w_2 ... w_n$, l'inversa e' la stringa $w^R = w_n, \cdots, w_2, w_1$. Per ogni linguaggio regolare A, sia $A^R = \{w^R | w \in A\}$. Mostrare che se A e' regolare, allora lo e' anche A^R.
2. Sia A/b $= \{w \mid bx \in A\}$. Mostrare che se A e' un linguaggio regolare e $b \in \sigma$, allora A/b e' regolare

1) Considerando il linguaggio A come regolare, in quanto composto da una serie di stringhe del tipo
$w = w_1, w_2, \dots w_n$.
Scegliendo nell'ipotesi del PL, una pumping length generica *h* avremo:
$w=w^k$
È intuitivo capire che qualsiasi combinazione di caratteri andiamo ad utilizzare data una qualsiasi suddivisione scelta di xyz, rispetta le solite condizioni (y ≠ε, |xy| <=k) ed il linguaggio rimane sempre regolare, per tutti gli esponenti "i" scelti per pompare la parola nella suddivisione xy^iz.
Stiamo parlando quindi di un esempio del tipo:

a….a a…a a…a
x y z

L'inversa non fa altro che applicare lo stesso principio alla rovescia, avendo infatti:
$w = w_n, w_{n-1}, \dots w_1$.
Quindi data sempre una parola
$w=w^k$
e una suddivisione

a….a a…a a…a
x y z

Non cambia assolutamente niente, entrambi i linguaggi sono considerabili regolari.

2) Assumiamo che L sia regolare e usiamo le solite y ≠ε, |xy| <=k
Come si vede A è regolare, in quanto formato da w.
Considerando proprio come parola una generica w=xyz, consideriamo ad esempio x=ε, $y=x^i$, z=ε
e si vede subito che qualsiasi pumping è regolare.
A queste condizioni, avremo che A/b è regolare, dato che anche b è formato letteralmente da parole del tipo $w=bx^i$. Si vede infatti che pompando i a piacere, rimane sempre verificata la proprietà |xy| <= z, con una sola b e x pompato, z sarà sempre formato da un numero da un esponente "n-i", rispettando la regolarità.
Quindi entrambi sono regolari.

2 Pumping Lemma

Dimostrare che i seguenti linguaggi non sono regolari

1. $\{0^n 1^m 0^n \mid m, n \geq 0\}$
2. $\{w \in \{0,1\}^* \mid w$ e' palindroma$\}$
3. $\{w \in \{0,1\}^* \mid w$ non e' palindroma$\}$
4. $\{0^{pq} \mid p, q \in \mathbb{N}, p > 1, q > 1\}$
5. $\{0^n 1^m 0^{m+n} \mid n, m \in \mathbb{N}\}$
6. $\{0^n 0^{2^n} \mid n, m \in \mathbb{N}\}$

1)

1) Lo scopo: dimostrare che non è regolare
A è regolare → ∃ k costante di pumping

2) Scelta parola w, |w| >= k
n = k
w = $0^n\ 1^m\ 0^k$
La parola deve essere almeno k, per regola, ma può essere anche oltre.

3) Suddivisione in xyz
w = xyz = 0……0|$1^n 0^k$
xy z
|y| > 0 $0^b 0^z$, a > 0
4) Scelta di i
xy^iz ∃A ∀i

Caso di esempio:
i=0
xz=$0^b 1^m 0^k$

000 1111000
xy z
Quindi posso scegliere in xy, la y, dato che so che blocco è, ad esempio quello evidenziato.
OCCHIO: Non scrivere xyz, cioè xy^iz all'esame

2)
1) Per assurdo, esiste l
2) scelta w
w=$0^k 1^k 0^k$

3) scelte xyz
w = 0…. 0 $1^k 0^k$
xy z
|y| = p, p > 0
|x| = k - p
w=$0^{k-p} 1^h 0^k$
xy^iz i=0 $0^{k-p} 1^h 0^k$

3)
Ragioniamo con il complementare (indicato da me per praticità ~A)
Quindi
L(~A) -> L(A)

L(A) reg -> L(~A) reg
Non è regolare quindi, dato che il complemento è anch'esso non regolare

4)
Per assurdo ammettiamo che esista L.
Scegliamo w, |w| >= k, q=k, |xy|< k, $z=0^{k-p}$
w = $0^{k-p}0^k$
xy^iz i=0, 0^{2k-p}
Di fatto come al solito |xy| non è <= k e quindi non è regolare

5) $0^n0^m0^{m+n}$
Per assurdo ammettiamo che esista L.
Scegliamo w, |w| >= k, n=k
$w=0^k0^m0^{m+k}$
xy^iz i=2, $0^k0^{2m}0^{m+k}$
$x=0^k$ $y=0^{2m}$ $z=0^{m+k}$
Basterà intuitivamente dare un qualsiasi valore ad i diverso da 1, quindi 0 o maggiore di 1 come banalmente 2 per dimostrare che la stringa pompata non è regolare.

6) $0^n0^{2^n}$
Per assurdo ammettiamo che esista L.
Scegliamo w, |w| >= k, n=k
$w=0^k0^{2^k}$
Un esempio banale; con i=0 si dimostra che il linguaggio non è regolare.
Infatti, assumendo:
$xy=0^k$ $z=0^{2^k}$
avremo $w=10^{2^k}$
che chiaramente non rispetta xy <= k

3 Lunghezza minima pumping

Per ognuno dei seguenti linguaggi, dare la lunghezza minima del pumping e giustificare la risposta.

1. 0001*
2. 0*1*
3. 1011
4. 1*01*01*
5. 001 + 0*1*

Quindi noi, scegliendo un opportuno valore di y, scegliamo uno "i" che permette di rispettare il PL.

1) ∃ K
   00011|1
   0001|1
   Necessariamente per condizione devo avere almeno:
   0001
   Devo quindi prendere y e poterla pompare, per realizzare le condizioni del PL
2) ∃ K
   Tale che partendo idealmente da k=2 avremo
   01
   Dunque, già la stringa risulta pompabile
3) ∃ K
   La lunghezza di k deve essere 5, in quanto con 4 assumeremmo che 1011 sia pompabile, quando essendo l'unica parola del linguaggio così non è
4) ∃ K

La lunghezza di k deve essere 5 anche qui, per poter avere almeno tutti caratteri diversi e sbilanciati da poter pompare

5) 0[01] 00101 non fanno parte della richiesta, in quanto violo tutta la condizione
Intuitivamente basta avere una lunghezza 2, a seguito di questa condizione risulta certamente pompabile.

Vediamo alcuni casi di esempio della minimum pumping length come:

18) il linguaggio vuoto
Dovrà essere 1 in quanto il numero di stringhe deve essere sempre maggiore di 0.
19) (01)*
Qua parliamo di 2, quindi partendo da 01.
20) 10(11*0)*0
Si direbbe 4, in quanto con meno cifre potenzialmente sarebbe sempre regolare il linguaggio, dato che nelle combinazioni dei numeri si potrebbe avere un eventuale ripetizione dei numeri che manterrebbero quindi bilanciata la parola.
21) 001 U 0*1*
La lunghezza minima può essere 1, considerando di avere la stringa vuota da una parte e 0 oppure 1 dall'altra.

L={$a^n b^{n+1}$]
Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: y diverso da 0, xy <= p e $xy^iz \in L$
Se poniamo come lunghezza di pumping h, poniamo poi h=n
e avremo: $a^p a^{p+1}$
Quindi la lunghezza 2p+1 >= p ed il linguaggio non è regolare.

L={$a^n b^l$ | n <= l)
Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: y diverso da 0, xy <= p e $xy^iz \in L$
Se avessimo quindi per ipotesi n=p
w=$a^p b^l$, con p<=l
Sapendo che p può essere anche uguale ad l, ciò riporta nelle condizioni
di un noto linguaggio non regolare.
Potremmo infatti trovarci nella situazione 2p >= p, dimostrando che in effetti non è regolare.

L={$a^k w$ | w ∈ {a, b}*, |w| = k)
Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: y diverso da 0 , xy <= p e $xy^iz \in L$
sapendo che w=k e dando un generico k, come ad esempio 2:
w=$a^2$2, riportandosi nella generica situazione: w=a^kk
Di fatto si vede che con qualsiasi k pompando non è regolare.

6. Let $A = \{1^j z | z \in \{0,1\}$ and z contains at most j 1's, for any $j \geq 1\}$. Prove, by the pumping lemma, that A is not regular.

Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: y diverso da 0 , xy <= p e $xy^iz \in L$
Dando una rappresentazione della stringa come xy^0z avremo:
x = ε, y=1^j z=z
In questo singolo caso avremmo proprio z come stringa e il linguaggio rimarrebbe regolare, essendo comunque < k, ma contenendo infatti j=0 uni.

Se invece avessimo ad esempio w=xy^2z

in questo caso per esempio sarebbe $w=1^j z^{k-j}$
in quanto consideriamo il caso in cui z sia composto da un esponente che considera le stringhe rimanenti.
Da qui si può già notare che in effetti pompando 1^j, otterremmo un numero di 1 che andrebbe bene (z che ha al più n uni) ma allo stesso tempo otterremmo un possibile sbilanciamento della stringa e come sempre |xy| <= k non è valida e il linguaggio non è regolare.

Use the pumping lemma to show that the following languages are not regular.

a. $A_1 = \{0^n1^n2^n \mid n \geq 0\}$

b. $A_2 = \{www \mid w \in \{a, b\}^*\}$

c. $A_3 = \{a^{2^n} \mid n \geq 0\}$ (Here, a^{2^n} means a string of 2^n a's.)

a) Il linguaggio, immaginando di rispettare le solite condizioni di PL, può essere facilmente considerato non regolare. Questo perché si nota che possiamo pompare finché vogliamo il numero di 1, sbilanciando il resto della stringa. Mentre 0 e 2 rimangono sempre con un numero n-esimo di cifre, gli 1 vanno per conto proprio ed otterremmo una cifra diversa da quella di partenza.
Si ha anche il caso interessante per cui y può essere composto da più numeri (es. 0011).
A quel punto sempre, pompando come prima per una generica xy^iz, si avrebbe una parola sbilanciata.
Dunque il linguaggio non è regolare.

b) Vogliamo dimostrare che il linguaggio sia non regolare. Presupponendo infatti una generica ripartizione di stringhe tra a e b, si nota che noi possiamo scegliere nel pattern due stringhe che si ripetono tra le 3 w, dimostrando ad esempio una suddivisione del tipo xy^iz in cui si ha:
$x=a^ja^ib$ si potrebbero avere parole come aaaaaaaabbbbb
oppure
$x=a^jb^ia$ si potrebbero avere parole come aaaabbaabbbaabbb
chiaramente combinazione sbilanciate.
Pertanto il linguaggio non è regolare.

c)

A_3 recognizes a, aa, aaaa, aaaaaaaa, ...

Choose *x* = a, *y* = aa, *z* = a seems a good idea, because we're sure that every string recognized by A_3 is writable with an xy^iz, but there are two problems: a single "a" is not writable with this pattern, and there are strings that are writable but don't be recognized by A_3, for example, i = 2 ⇒ a aa aa a (the spaces have been added to make the string more readable).

Choosing a simpler definition of *x*, *y*, *z*, as *x* = ε, *y* = a, *z* = ε, we are affected by the same problem.

8. Sia $\Sigma = \{0, 1\}$.

- Mostrare che il linguaggio $A = \{0^k u 0^k \mid k \geq 1 \text{ e } u \in \Sigma^*\}$ è regolare.
- Mostrare che il linguaggio $B = \{0^k 1 u 0^k \mid k \geq 1 \text{ e } u \in \Sigma^*\}$ non è regolare.

22) Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: y ≠ 0 , xy <= p e $xy^iz \in L$
Si nota subito che dando come stringa xy^0z abbiamo subito 0^k0^k, tale da avere già una stringa regolare. Anche negli altri casi, pompando genericamente la stringa con un indice i, notiamo che il numero di zero rimane uguale e il linguaggio rimane regolare.

23) Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: y ≠ 0 , xy <= p e $xy^iz \in L$
Diversamente da prima, potrebbe benissimo esserci il caso in cui:
$x=0^k$ $y=1u$ $z=0^k$ tale per cui rimane regolare

Tutta via può esserci il caso in cui si abbia una diversa suddivisione, tipo:
$x=0^k1$ $y=u$ $z=0^k$
per cui la stringa può essere sbilanciata e il linguaggio non sarà regolare.

7. Sia $\Sigma = \{0, 1\}$, e considerate il linguaggio

$$D = \{w \mid w \text{ contiene un ugual numero di occorrenze di 01 e di 10}\}$$

Mostrare che D è un linguaggio regolare.

Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: $y \neq 0$, $xy <= p$ e $xy^iz \in L$
Consideriamo come parola $w=(01)^n(10)^n$
Sappiamo quindi che esisteranno due esponenti, $k > 0$, $j > 0$ tale che $(01)^k(10)^j$ entrambi minori di k, con $x=\varepsilon$.
Chiaro è, a queste condizioni, che per qualsiasi i tale da pompare y, avremo $(01)^{k+i}(10)^{n-j-k}$ tale che il numero di occorrenze di 01 ed 10 si pareggia per ogni possibile pumping length.
Quindi il linguaggio è sempre regolare.

$L = \{0^n1^m2^n \mid n, m \geq 0\}$

Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: $y \neq 0$, $xy <= p$ e $xy^iz \in L$
Scegliamo come possibile parola $w=0^k12^k$
Si può vedere quindi che i pumping su 1 portano il linguaggio ad essere sbilanciato, infatti già da qui avremmo un numero di esponenti $2p+1 > p$. Infatti seguendo la classica scia di dimostrazione, si noterebbe che una possibile stringa z avrebbe ad esempio $2^{p-(2n+1)}$ che a seguito di pumping sarebbe chiaramente sbilanciata. Pertanto il linguaggio non è regolare.

$L = \{0^k10^k \mid k \geq 0\}$

Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: $y \neq 0$, $xy <= p$ e $xy^iz \in L$
Scegliamo come possibile parola $w=0^n10^n$
Il numero di 0 è chiaramente sbilanciato rispetto agli 1, avendo una lunghezza $2n > p$ e si ragiona ugualmente a prima (z che ragiona per sottrazione di "p" dal resto degli esponenti).
A seguito di pumping si avrebbe chiaro sbilanciamento; pertanto il linguaggio non è regolare.

$L = \{a^ib^n \mid i, n \geq 0, i = n \text{ or } i = 2n\}$

Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: $y \neq 0$, $xy <= p$ e $xy^iz \in L$
$w=a^pb^p$. Eseguendo il pumping su uno dei due pezzi, ad esempio su a^p
per p=2 avremmo tipo:
$a^{2p}b^p$ che dimostra che abbiamo un numero diverso di a rispetto alle p nella parola e il linguaggio non può essere regolare.

$L = \{a^nb^lc^k \mid k \neq n + l\}$

Assume L is regular. From the pumping lemma there exists a p such that every $w \in L$ such that $|w| \geq p$ can be represented as x y z with $|y| \neq 0$ and $|xy| \leq p$. Let us choose $a^{p!}b^{p!}a^{(p+1)!}$. Its length is $2p!+(p+1)! \geq p$. Since the length of xy cannot exceed p, y must be of the form a^m for some $m > 0$. From the pumping lemma any string of the form xy^iz must always be in L. If we can show that it is always possible to choose i in such a way that we will have $k = n + l$ for one such string we will have shown a contradiction. Indeed we can have

$$p!+(i-1)m + p! = (p+1)!$$

if we have $i = 1 + ((p+1)! - 2\,p!)/ m$ Is that possible? only if m divides $((p+1)! -2\,p!$

$((p + 1)! - 2 * (p)! = (p + 1 - 2)\,p!$ and since $m \leq p$ m is guaranteed to divide p!.

Hence i exists and the language is not regular.

$L = \{ a^{n!} \mid n \geq 0\}$
Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: $y \neq 0$, $xy <= p$ e $xy^iz \in L$
Assumiamo quindi di avere $a^{i!}$ per $i > 0$
In queste condizioni, avremo una possibile suddivisione che considera una stringa ripetuta con potenza "k" >= 1, quindi avremo $a^{i!-k}$
In questo caso quindi $p = i! – k$.
Nel momento in cui abbiamo un pumping della stringa, tipo con $k > 2$, risulta che $i! – k > (m – 1)!$ che dimostra la non regolarità.

$L = \{va^{k+1} \mid v \in \{a, b\}^*, |v| = k\}$
Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: $y \neq 0$, $xy <= p$ e $xy^iz \in L$
Ipotizziamo una parola $w=b^ka^{k+1}$ tale che sia nel linguaggio
Se immaginiamo un pumping qualsiasi, ad esempio per "i" avremo la situazione di avere una cosa del tipo $w=b^{i+k}a^{k+1}$ che dovrebbe rimanere nel linguaggio.
In questo caso però non viene rispettata $|xy| <= k$ e infatti avremmo che ad esempio con $b^{3+k}a^{k+1}$ non troveremmo posto nel pattern.
Infatti $3+k > k$ per qualsiasi pumping eseguito e il numero di "b" sbilancia quello delle "a".

$L = \{va^{2k} \mid v \in \{a, b\}^*, |v| = k\}$
Per assurdo come sempre assumiamo L sia regolare.
Quindi dovremmo avere sempre: $y \neq 0$, $xy <= p$ e $xy^iz \in L$
Prendiamo quindi una parola b^pa^{2k} con $p > 0$
Quindi se eseguiamo un generico pumping "i" su b potremmo avere ad esempio:
$b^{p+i}a^{2k}$ ad esempio quindi $b^{3p}a^{2k}$
Se sappiamo che b^ka^p è nel linguaggio, anche $b^{k+i}a^p$ dovrebbe essere nel linguaggio, ma abbiamo anche che $v=k$ quindi avremmo: b^ka^{2k} con un numero doppio alla k di a rispetto a b.
Basta appunto eseguire il pumping ed il linguaggio è sbilanciato.

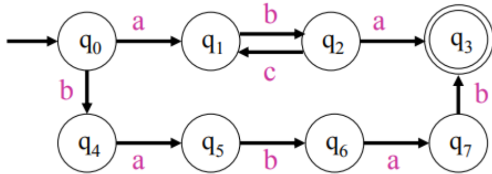
$L_2 = a(bc)^*ba$

la stringa "aba" non può essere suddivisa secondo le regole del pumping lemma; ogni stringa di lunghezza > 3 invece è del tipo "abcbc..bcba" e può sempre essere suddivisa al modo u = "a" v="bc" e w = parte restante; allora basta scegliere *n* = 4 (o *n*=5) affinché siano verificate le proprietà del pumping lemma (osserva che non possono esistere stringhe di lunghezza pari nel linguaggio)

esempio: z = abcbcba $\qquad$ z_3 = abcbcbcbcba

E' facile ricavare un ASF con 4 stati (escluso lo stato pozzo) che riconosce il linguaggio L_2

soluzione esercizio 1.c L_3= a(bc)*ba + babab
risulta $L_3 = L_2 \cup \{babab\}$, quindi per tutte le stringhe del linguaggio, tranne che per la stringa “babab”, si può ragionare come per il linguaggio L_2;
tuttavia, la stringa “babab” non può essere suddivisa secondo le regole del pumping lemma, ed ha lunghezza 5;
quindi occorre scegliere $n \geq 6$



soluzione esercizio 3 ($L = \{a^h b^k c^{h+k} : h, k > 0\}$ non regolare)
è possibile utilizzare entrambe le tecniche (debole o con utilizzo di tutte le ipotesi) per negare il pumping lemma:

• utilizzo della tecnica debole (mostro che non posso mai suddividere)
  – sia uvw ($|v| \geq 1$) una suddivisione per la stringa $z = a^h b^k c^{h+k}$;
  – v non può essere fatta di sole ‘a’, perché altrimenti “pompando” v si avrebbe solo una variazione del numero di ‘a’, mentre il numero di ‘b’ e di ‘c’ rimarrebbe uguale (sbilanciamento)
  – analogamente a sopra, v non può essere fatta di sole ‘b’ o di sole ‘c’ (sbilanciamento)
  – infine, v non può prendere simboli misti, perché altrimenti si avrebbero delle alternanze

24

Dimostrare che il linguaggio $\{a^n b^m a^n | m \geq n\}$ non è CF.

Se un linguaggio non è context-free allora non è regolare e, per non essere regolare, bisogna applicare il solito pumping lemma. Si può subito notare come il numero di a sia almeno “2n” con il numero di b che è “m”. Tuttavia, la proprietà asserisce che m >=n e dunque dovremmo avere, per le solite condizioni:
$y \neq 0$, $xy <= p$ e $xy^iz \in L$
una parola $w=a^{2k}b^m$ tale che sia $w=a^{2k}b^{p-m-2k}$. Sapendo che m > 2k, necessariamente la parola contiene più a che b nella sottostringa “xy” e dunque il linguaggio non è regolare e dunque non potrà essere CF

Esercizi Grammatiche context-free

Dal file "05-esercizi.pdf"

Esercizio 1

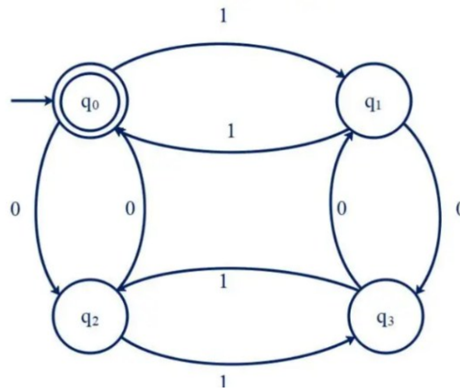
Definire una grammatica per il seguente linguaggio:

$$L_1 = \{a^i b^j c^k \mid i = j \text{ oppure } j = k\}$$

In pratica si ragiona che, avendo "a" e "b" che si incrementano per lo stesso indice, avendo poi c, di fatto l'idea è di garantire l'alternanza tra "a" e "b", poi unendo l'altro caso dell'oppure, dove si ha lo stesso discorso ma tra "b" e "c".

Per il caso i=j	*Per il caso j=k*
$S_1 \rightarrow aS_2bS_3 \mid \varepsilon$	$S_1 \rightarrow bS_2cS_3 \mid \varepsilon$
$S_2 \rightarrow aS_2b \mid \varepsilon$	$S_2 \rightarrow cS_2 \mid \varepsilon$
$S_3 \rightarrow bS_3c \mid \varepsilon$	$S_3 \rightarrow bS_3c \mid \varepsilon$

Automa corrispondente:



Oppure:

Answer: $G = (V, \Sigma, R, S)$ with set of variables $V = \{S, W, X, Y, Z\}$, where S is the start variable; set of terminals $\Sigma = \{a, b, c\}$; and rules

$$\begin{aligned} S &\rightarrow XY \mid W \\ X &\rightarrow aXb \mid \varepsilon \\ Y &\rightarrow cY \mid \varepsilon \\ W &\rightarrow aWc \mid Z \\ Z &\rightarrow bZ \mid \varepsilon \end{aligned}$$

Esercizio 2

Definire una grammatica per il seguente linguaggio:

$$L_1 = \{w \in \{0,1\}^* \mid w \text{ contiene un numero pari di 0 e un numero dispari di 1}\}$$

$S_1 \rightarrow 1S_1 \mid S_11 \mid \varepsilon$
$S_2 \rightarrow 0S_2 \mid S_20$
$S_3 \rightarrow 1S_3 \mid S_31$
$S_4 \rightarrow 0S_4 \mid S_40 \mid \varepsilon$

Esercizio 3

Definire una grammatica per il linguaggio di tutte le espressioni regolari sull'alfabeto $\{0, 1\}$, cioè di tutte le espressioni costruite utilizzando

- le costanti di base: $\varepsilon, \emptyset, 0, 1$
- gli operatori: $+, \cdot, *$
- le parentesi

FACTOR → (EXPR) | Ø | ε | 0 | 1
(EXPR) → <FACTOR>
SUM → SUM + SUM | PRODUCT
PRODUCT → PRODUCT * PRODUCT | STAR
STAR → FACTOR* | FACTOR

a) $L = a^i b^j c^k | i = j + k\}$

Si ragiona che si può ripetere "a" oppure "c", conseguentemente cambia l'intermezzo:
S_1 → $aS_1c|S_2$
S_2 → $aS_2b|\varepsilon$

b) $L = \{a^i b^j c^k | j = i + k$

Simile ma diverso da prima, nel senso che ora è l'indice intermedio a cambiare. Si devono quindi considerare tutti i casi di shift e ripetizione tra le parti.
S_1 → $aS_1bS_2|S_1bS_2c|\varepsilon$
S_2 → $aS_2b|\varepsilon$
S_3 → $bS_3c|\varepsilon$

(a) $\{ w \in \{0,1\}^* \mid w$ contains at least three 1s $\}$

La stringa contiene almeno tre 1 e ricorsivamente potrà avere 0/1/stringa vuota.
S_1 → $S_1 1 S_1 1 S_1 1 S_1$
S_2 → $0S_1|1S_1|\ \varepsilon$

(b) $\{ w \in \{0,1\}^* \mid w = w^{\mathcal{R}}$ and $|w|$ is even $\}$

S_1 → $0S_10 \mid 1S_11 \mid \varepsilon$

- Create an CFG for $L = \{a^i b^j c^k : j > i + k\}$

S → ABC
A → aAb | ε
B → bB | b
C → bCc | ε

4. Give a context-free grammar to generate the language $(0+1)0^*(11)^*$.

S → ABC
A → 0|1
B → 0B|ε
C → 11C | ε

(c) $\{ w \in \{0,1\}^* \mid$ the length of w is odd and the middle symbol is 0 $\}$

Essendo dispari potrà avere il caso pari più almeno un simbolo diverso per essere dispari, essere già dispari oppure avere appunto 0 come simbolo di mezzo nella stessa stringa.
S_1 → $0S_10 \mid 1S_11 \mid 0 \mid 0S_11 \mid 1S_10$

What is the language generated by $G_1 = (\{a, b, S, T, U\}, \{a, b\}, S, P)$ if P is altered to:

$$S \to T \qquad S \to bSb \qquad T \to aT \qquad T \to \epsilon$$

bSb – bbSbb – bbTbb – bbaTbb – bbaεbb (esempio di stringa generabile)
Quindi il linguaggio potrebbe essere b*ab*

Given language $A = \{a, b, ab\}$ and $B = \{c, d, cd\}$. What is AB? What is A^*? What is $\{a, b\}^*$?

AB = aR1|bR1|abR1|cR2|dR2|cdR2
A* = ε|a|b|ab
{a,b}* = ε|a|b|ab

Find a context-free grammar that generates the set of all palindromes over the alphabet $\{0, 1\}$

S → 0A0 | 1S1 | ε
A → 0B0 | 1A1
B → 1B1 | 0S0 | ε

Esercizio 5.1.1 (d). Vogliamo definire una CFG che genera il linguaggio L composto da tutte le stringhe in {a,b}* tali che hanno un numero di b doppio rispetto agli a.

Definiamo la grammatica nel seguente modo:
S → SaSbSb | SbSaSb | SbSbSb

1)Data la seguente grammatica libera da contesto G : S ← aS | aSbS | ε, dimostrare che il linguaggio L(G) contiene solo stringhe tali che ogni loro prefisso abbia un numero di a almeno pari al numero dei b.

Induttivamente, consideriamo che la grammatica appena descritta porti ad una serie di derivazioni successivi del tipo S → aS → aaS → aaaSbS e così via. Nel caso facessi S → aSbS → aaSbSbaSbS partendo come una non leftmost derivation, si può vedere che il numero degli b è al più <= al numero di a e, infatti, ogni pezzo/prefisso della derivazione attuata porta a questo tipo di derivazione.

Costruire una CFG G che genera il linguaggio $L = \{a^n b^m c^k \mid \text{ con } n = m \text{ o } m = k \text{ e } n, m \text{ e } k \geq 0\}$. Dimostrare che per la grammatica G che proponete, vale $L(G) \subseteq L$.

In questa grammatica considero che le stringhe siano del tipo → aabbc oppure abbcc
Quindi avremo un linguaggio del tipo:
S → AB | BD
A → aAb | ε
C → cC | ε
B → aB | ε
D → bDc | ε

Let L be the language $\{w \in \{a, b\}^* | w$ contains exactly one more b than $a\}$.

(a) Give a context-free grammar that generates L.

S → aB | bA | ε
A → aS | bAA
B → aBB | bS

oppure

S → TbT
T → aTb | bTa | TT | ε

Tutorato 4

Ex 1 Grammatiche libere da contesto

Definire delle gramatiche libere da contesto per i seguenti linguaggi

A → AB|C
Le regole si applicano solo a variabili e quindi si ha la conclusione.

Ex 1.2

$\{w \mid w = w^R$,w e' palindroma$\}$

Σ={0,1}
w= 001 100
x x^R
R → 0R0|1R1|1|0|ε
R → 0R0 → 01R10 → 0110

L'ordine delle variabili nell'automa, ma nelle grmamatiche è molto libero, richiamando senza neessariamente un ordine.
E.g. S → ABC
A → BA
C → d
B → x

Ex 1.1

{ w | la lunghezza di w e' dispari con all'interno il simbolo zero}

First try (non va bene, dato che non rispettiamo i vincoli sulla lunghezza della stringa.
w = (0+1)*0(0+1)* produrrebbe 1101 che non va bene

w = sx 0 dx
|w| = |sx| + 0 + |dx|

-pari + pari = pari
-dispari + dispari = pari
-pari + dispari = dispari

S → P0P|D0D
P → 00P|01P|10P|11|ε
D → 0|1|0P|1P|ε

Ex 1.3

$\{w\#x \mid w^R$ e' una sottostringa di x $\}$

L'idea è di produrre "ad alto livello" tutte le rappresentazioni di 0/1.
Cerchiamo di formare dei blocchi elementari, ricombinati insieme.

w=100
x=(0+1)*001(0+1)*
w#(0+1)*w^R(0+1)

X → 0X|1X|ε
Un'idea può essere di avere una rappresentazione riflessa, in modo simmentrico:
X → 0X|1X|ε
T→0T0|1T1|ε

Se io decidessi di rappresentare le altre stringhe avrei richiamdno tutto:
T → 0T0|1T1|#X
T → 0T0 → 01T10 → 01#X10

Qui si interpretano le regole in modo non lineare, con la X chiamata 2 volte:
S → TX
T → 0T0|1T1|#X
X → 0X|1X|ε

$\{x\#y \mid x, y \in \{0,1\}^{\star}, x \neq y\}$

Ragionando che tutti i simboli devono essere diversi, stando ovviamente tra 0/1, abbiamo che ciascuno di questi alterna gli uni con gli in maniera regolare. Lo stato iniziale pone # tra l'alternanza delle due stringhe, avendo poi A che decide di alternare A con 0 e 1 oppure può essere 1 semplicemente, garantendo alternanza con B che si comporta in maniera totalmente duale. Segue poi tutto il resto.

S → A#B|B#A|ε
A → TAT|0
B → TBT|1
T → 0|1

Ex 2 Grammatiche ambigue

Considerate la seguente grammatica

S → A | If-then | If-then-else
If-then → if cond then S
If-then-else → if cond then S else
A → a:=1

Ex1 Dimostrare che la grammatica e' ambigua
Ex2 Modificare la grammatica per rimuovere l'ambiguita'

Esempio di grammatica ambigua (posso disambiguare semplicemente cambiando l'ordine di derivazione, chiamando prima "c"):
A → BC A → BC → bC → bc
B → b A → BC → Bc → bc
C → c

Possibili esempi:
S → If-then → If-cond-then-S → If-cond-then-If-then-else→If-cond-then-IF-cond-S-else-S

S → If-then-else → If-cond-then-S-else-S → If-cond-then→ If-cond-then-If-cond-else-S

Non è una parola in quanto dovrebbe essere terminale per essere considerata tale.
L'idea sintattica sarebbe:

IF(cond A){
if(cond B) statement else
statement

}

IF(cond A){
if(cond B) statement A
}
else statement B

Quindi avere un classico blocco if-then oppure avere due blocchi if-else con degli if appaiati (quindi innestati) al primo.
Per rimuovere l'ambiguità dobbiamo modificare le regole e quindi ne aggiungiamo due nuove:
Open, Closed
Open può essere con IF singolo, Closed non ha If oppure ogni If ha il suo else

L'idea è di avere degli "if" ambigui, che potrebbero collegarsi prima/dopo con altri if oppure con altri blocchi.
If ← If → else
If then S else S
If then If then else S
(qui posso aggiungere altri if, dato che questo if sarebbe interno e dunque non più ambiguo)

S → Open |Closed
Open → If-Cond-then-S|If-Cond-Then-Closed|else-Open (qui sarebbe ambiguo perché non si saprebbe come interpretarlo)
Closed → A|If-Cond-Closed-else-Closed

Qui potrebbe rinascere l'ambiguità:

A → a:=1
S → Open→If-cond-then-S→If-cond-then-Closed
(l'idea è che Closed in sé contribuisce alla chiusura di tutti i costrutti).

Ex 3 Forma normali

Sappiamo che la regola è:
S → A
A → BC
A → B
S → ε

L'obiettivo è di convertire la seguente grammatica:

$A \rightarrow BAB|B|\epsilon$
$B \rightarrow 00|\epsilon$

La variabile iniziale appare a dx e quindi introduciamo la variabile iniziale

Passo 1: nuovo stato iniziale
S→A
A→BAB|B|ε
B→00|ε

Passo 2: rimuovere ε
Rimuovo A→ε

S→A|ε
A→BAB|B|BB
B→00|ε

Rimuovo B→ε
S→A|ε
A→BAB|B
B→00|ε

BAB, AB, BA, A

S→A|ε
A→BAB|B|BA|A|AB|BB
B→00

Passo 3: elimino regole unitarie
Quindi si avrà da togliere A → A, S → A, A → B
Ovviamente posso togliere senza problemi A→A:

S→A|ε
A→BAB|B|BA|AB|BB
B→00

Poi si toglie S → A

S→BAB|B|BA|AB|BB|ε
A→BAB|B|BA|AB|BB
B→00

E infine si toglie A → B

S→BAB|B|BA|AB|BB|ε
A→BAB|00|BA|AB|BB
B→00

Ora dobbiamo togliere le transizioni eccedenti che producono almeno 2 simboli non terminali.
Vediamo ad esempio S → BAB, A → BAB
Possiamo inserire ad esempio come regola Q → AB

S→BQ|B|BA|Q|BB|ε
A→BQ|00|BA|Q|BB
B→00
Q →AB

Notiamo ora che BA risulterebbe terminale e lo sostuiamo con regola apposita R → BA

S→BQ|B|R|Q|BB|ε
A→BQ|00|R|Q|BB
B→00
Q →AB
R → BA

Non essendo più rimaste cose terminali, togliamo la ε ed otterremmo:

S→BQ|B|R|Q|BB
A→BQ|00|R|Q|BB
B→00
Q →AB
R → BA

Convertire in forma normale le seguenti grammatiche

Ex 4.1

S → aXbX
X → aY|bYϵ
Y → X|c

Ex 4.2

S → AbA
A → Aa|ϵ

4.1)

Non occorre aggiungere un nuovo stato iniziale dato che non compare a dx.
Dobbiamo però rimuovere le ε-transizioni, nello specifico
- togliamo X → ε (tra bY ed ε occorrerebbe | nella scrittura)
S→ab|aXbX
X→aY|bY
Y→X|c

Ora dobbiamo rimuovere le regole unitarie, nello specifico:
Y → X Y→c
nel primo caso sarà:
S→ab|aXbX
X→aY|bY
Y→ aY|bY|c

Per la regola terminale unitaria Y→c occorrerà aggiungere una nuova regola terminale (anche perché servono solo due variabili, notando che siano diverse per forma tipo BA/AB sono la stessa variabile in pratica, a destra), tipo:
S→ab|aXbX
X→aY|bY
Y→ aY|bY|C
C → c

4.2) Anche in questo caso non occorre introdurre un nuovo stato iniziale.
Occorre prima di tutto eliminare la ε**,** quindi A → ε
S→Ab|bA|b|AbA
A→Aa|a

Occorre quindi rimuovere le due regole unitarie presenti S→b e A→a
Essendo entrambe regole terminali, dobbiamo introdurre due nuove regole.
Inoltre notiamo che in S compare due volte bA; si può direttamente sostituire con una nuova regola.

S→Ab|C|D|AC

A→AE|E
C→bA
D→b
E→a
In questo modo abbiamo concluso la normalizzazione.

(c) $\{\ 0^i1^j \mid i \leq j \leq 2i\ \}$

Sappiamo che il numero di 1 è doppio al numero di 0, inoltre nel secondo passaggio si ragiona che a parità di 0 avrò lo stesso numero di 1 per la disuguaglianza, oppure semplicemente non avere nessun numero.
Quindi:
S→0S11|A
A→0A1|ε

$\{\ x \in \{0,1\}^* \mid \neg(\ (\exists i,j)(x = (0^i1^i)^j)\ )\ \}$

Esempio interessante, in cui si argomenta che il linguaggio accetta delle stringhe in cui non esistono delle "i" e "j" per poter dire che x presenta un numero di 0 e di 1 elevati alla i tutto elevato alla j.
Quindi un esempio di stringhe non accettabili è (rispettando l'ordine di ideazione):
- stringhe che iniziano con 0
- stringhe che finiscono con 1
- numero uguale di 0/1 nelle sequenze 0^n1^n oppure 1^n0^n

A queste condizioni proviamo a definire la grammatica step by step, partendo da A che accetta tutto:
A → 1A0|0A1|ε

Ora definiamo il simbolo iniziale tale da violare i discorsi "iniziano con 0" oppure "finiscono con 1":
S → A0|1A

Dobbiamo quindi gradualmente aggiungere regole per fare in modo mi generi solo le negazioni di quello che mi serve, quindi:
B→ 0B1 | 0C1 |0D1
C→ 0C|0
D→1D|1

a questo punto mi interesso della terza condizione, generabile nel caso in cui abbia una regola tipo:
E→0E1|1E0|ε
volendo spezzettabile in una serie di parti, come
S → EBF
E → 0E|0
F → F1|1

oppure analogamente per il caso inverso
G → 1G0 | 1H0 | 1J0
H →1H|1
J → 0J|0

Riunendo tutto insieme, si vede quindi che A dovrà avere un numero contrapposto di produzioni in un senso e nell'altro e per violarle tutte useremo G e B (con G che già incorpora B in se stessa):
S → A0G1A

For the grammar G with the productions:

```
S -> A | I | E
I -> if cond then S
E -> if cond then S else S
A -> a
```

Find a **leftmost** derivation for the word:

if cond then if cond then a else s

HELP: Derivation Syntax +

```
S
I
if cond then S
if cond then E
if cond then if cond then S else S
if cond then if cond then A else S
if cond then if cond then a else S
if cond then if cond then a else s
```

L'idea di grammatica è questa con la leftmost; dà sbagliato semplicemente perché manca una regola S → s, a quel punto sarebbe certamente giusta. Per come è formattato l'esercizio, darà sempre sbagliata, perché la sostituzione di E ha tutte e due le S maiuscole, ma l'esercizio richiede la "s" minuscola per passare, di cui non c'è nessuna regola.

Find a grammar that recognizes the following language:

a^i b^j c^k tali che i = j oppure j = k

```
S->X|Y
X->Xc|A|\eps
A->aAb|\eps
Y->aY|B|\eps
B->bBc|\eps
```

Find a grammar that recognizes the following language:

tutte le parole w che contengono un numero pari di 'a' e dispari di 'b'

```
S->aT|bR
T->aS|bU
R->bS|aU|_
U->aR|bT
```

What is Context Free Grammar CFG for language of all even length strings?

S ⇒ aSa | bSb | aSb | bSa |ε

- Example. Consider a CFG which generates the language consisting of even number of a's and even number of b's:

$S \to aB \mid bA \mid \lambda$ {S: even a's and even b's}
$A \to aC \mid bS$ {A: even a's and odd b's}
$B \to aS \mid bC$ {B: odd a's and even b's}
$C \to aA \mid bB$ {C: odd a's and odd b's}

- Example. Same as above except odd a's and odd b's

$S \to aB \mid bA$
$A \to aC \mid bS$
$B \to aS \mid bC$
$C \to aA \mid bB \mid \lambda$

```
L = { a^x b^y : y>=x, y-x is odd }
```

S → aSb | B
B → b | bbB

06b-esercizi – Grammatiche context-free

Considera l'alfabeto $\Sigma = \{0, 1\}$, e sia L_3 l'insieme di tutte le stringhe che contengono almeno un 1 nella loro seconda metà. Più precisamente, $L_3 = \{uv \mid u \in \Sigma^*, v \in \Sigma^*1\Sigma^* \text{ e } |u| \geq |v|\}$. Definisci una CFG che genera L_3.

$A \to 0A0 \mid 0B1 \mid 1A0 \mid 1B1$
$B \to 0B0 \mid 0B1 \mid 1B0 \mid 1B1 \mid 0 \mid 1 \mid \varepsilon$

Esercizio 4

Definire le grammatiche context-free che generano i seguenti linguaggi. Salvo quando specificato diversamente, l'alfabeto è $\Sigma = \{0, 1\}$.

1. $\{w \mid w$ contiene almeno tre simboli uguali a $1\}$
2. $\{w \mid$ la lunghezza di w è dispari$\}$
3. $\{w \mid w = w^R$, cioè w è palindroma$\}$
4. $\{w \mid w$ contiene un numero maggiore di 0 che di $1\}$
5. Il complemento di $\{0^n 1^n \mid n \geq 0\}$
6. Sull'alfabeto $\Sigma = \{0, 1, \#\}$, $\{w\#x \mid w^R$ è una sottostringa di x e $w, x \in \{0, 1\}^*\}$

1) S → X1X1X1X
X → 0X | 1X | ε

2) S ⇒ 0S0 | 1S1| 0S1 | 1S0 |ε

3) S → 0S0
S → 1S1
S → ε
S → 0

S → 1

4) S → TS|0T|0S
T → TT|1T0|0T1|ε

5)
S→1A|A0|0S1
A→0A|1A|ϵ

6)

$$\begin{array}{lcll} S & \rightarrow & CB & \triangleright \text{ Generates } w\#\Sigma^* w^R \Sigma^*. \\ C & \rightarrow & 0C0 \mid 1C1 \mid \#B & \triangleright \text{ Generates } w\#\Sigma^* w^R. \\ B & \rightarrow & AB \mid \varepsilon & \triangleright \text{ Generates } A^* = \Sigma^*. \\ A & \rightarrow & 0 \mid 1 & \triangleright \text{ Generates } \Sigma. \end{array}$$

Esercizi Forme normali di Chomsky/PDA

Dal file "06-formenormali.pdf"

Esercizio 1

Trasformare la seguente CFG in forma normale di Chomsky, usando l'algoritmo mostrato nelle slide:

$$A \rightarrow BAB \mid B \mid \varepsilon$$
$$B \rightarrow 00 \mid \varepsilon$$

Aggiungiamo la nuova variabile iniziale:
S_0 → A
A → BAB|B|ε
B → 00|ε

Rimuoviamo le ε-regole, quindi B → ε ed A → ε, partendo proprio da B → ε:
S_0 → A
A → AB|BA|B|BAB|ε
B → 00

e poi A → ε (che più di tanto non ha effetto).
S_0 → A
A → AB|BA|B|BAB|A
B → 00

Ora vogliamo togliere le regole unitarie, quindi A→B, A→A, S_0→A, partendo da A→A che semplicemente:
S_0 → A
A → BAB|B|AB|BA
B → 00

poi vogliamo togliere A→B (sostituendo B con 00):
S_0 → A
A → BAB|00|AB|BA
B → 00

quindi togliamo S_0→A (sostituendo BAB|00|AB|BA ad A):
S_0 → BAB|00|AB|BA
A → BAB|00|AB|BA
B → 00

Seguendo troviamo le produzioni che hanno più di 2 variabili a destra:
S_0 → BAB   A → BAB
introducendo una variabile generica X che sostituisce ad esempio BA:
S_0 → XB|00|AB|BA
A → XB|00|AB|BA
B → 00
X → BA

Notiamo inoltre 00 come simbolo terminale e gli andrà aggiunta una regola:
S_0 → XB|Y|AB|BA
A → XB|Y|AB|BA
B → 00
X → BA
Y → 00

Esercizio 2

Per ogni linguaggio L, sia

$$\text{suffix}(L) = \{v \mid uv \in L \text{ per qualche stringa } u\}.$$

Dimostra che se L è un linguaggio context-free, allora anche suffix(L) è un linguaggio context-free.

Suggerimento: puoi assumere che L sia generato da una grammatica in Forma Normale di Chomski.

Assumendo che il linguaggio sia definito secondo la seguente grammatica:
S → u
u → v | uv

dobbiamo ricondurci ad avere una cosa del tipo:
A → BC
A → a

A questo punto si nota che c'è una forma unitaria del tipo u → v, ma anche S → u. Una eventuale sostituzione del primo caso porterebbe a:
S → v | uv
u → v | uv

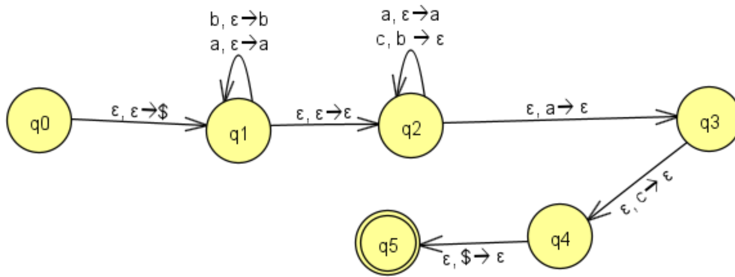
e dunque meglio tenersi:
S → u
u → v | uv

Meglio quindi introdurre una nuova regola, essendo "u" stato terminale, tale da sostituirlo:

S → X
u → v | uv
X → u

Tale linguaggio rientra nella forma normale definita da Chomsky.
A tale scopo si vede che la stringa è uniformemente ripartita tra u e v, nelle tre parti possibili.
Possiamo ipotizzare seguendo il PL una stringa del tipo $w=u^p v^p$ vedendo come è scritto il linguaggio.
Si nota subito che avendo ad esempio: $x=\varepsilon$, $y=u^k$, $z=v^j$ per qualche j, k > 0 si avrà una presenza normale di 0 ed 1 ed equamente distribuita. Inoltre, per la stringa di u^k sarà sempre previsto un numero di 0 che rientra nella condizione $|xy| <= k$, quindi del tipo $w=u^p v^{k-p}$, tale che qualsiasi pumping rientri nel linguaggio.
Essendo regolare, il linguaggio è context-free.

11) Descrivere un PDA che accetta per pila vuota ed è capace di riconoscere il linguaggio $L = \{(ab)^n(ca)^n \mid n \geq 1\}$. Il vostro è un automa deterministico o nondeterministico? Spiegare la risposta.



14. Dimostrare che se G è una CFG in forma normale di Chomsky, allora per ogni stringa $w \in L(G)$ di lunghezza $n \geq 1$, ogni derivazione di w richiede esattamente $2n - 1$ passi.

Sapendo che G è in forma normale di Chomsky, quindi nella forma
A→ BC
A → a

Prendiamo l'esempio di una grammatica di un es. fatto in questo file:
S' → AX|YB|a|AS|SA
S → AX|YB|a|AS|SA
A → b|AX|YB|a|AS|SA
B→ b
X → SA
Y → a

Consideriamo una stringa di lunghezza 1, possibilmente composta da una situazione del tipo
A → BC
Sono tutti simboli terminali e quindi, ciascuno è in forma normale di Chomsky, non avendo regole unitarie/transizioni vuote/più di tre transizioni per regola.
Induttivamente, nel caso di n > 1, avremo una situazione in cui, per poter applicare Chomsky abbiamo bisogno di almeno 2 regole, una terminale e non terminale ed eventuali altre regole.
Quindi:
A → BC
B → b
C → b|c|d
Ogni singola derivazione, per Chomsky, richiede esattamente l'applicazione di una sostituzione in altra regola e ogni regola può essere composta fino ad un massimo di 2 regole.
Essendoci almeno una regola terminale, per ipotesi, il numero di stringhe necessarie a comporre tale situazione sarà 2n.
Data appunto la presenza di almeno una regola terminale, il numero di stringhe necessarie a mostrare tale situazione in una derivazione si compone di $2n - 1$ passi.

In una situazione reale come la grammatica ottenuta sopra, si nota che per ogni simbolo di partenza ne corrispondono almeno 2 derivati, con l'aggiunta della regola terminale.
Consideriamo quindi S' → AB ed S → XY. Tale situazione comporta due regole non terminali ed un'altra possibilmente terminale. Dunque generalmente Chomsky richiede questo tipo di derivazione.

Convertire nella forma normale di Chomsky la seguente grammatica:
S → a|aA|B
A → aBB|ε
B → Aa|b

Il simbolo iniziale non appare a destra, possiamo quindi andare a togliere le ε-regole.
Esse sono A → ε
S → a|aA|B
A → aBB
B → a|Aa|b

Rimuoviamo la regole unitarie quindi S→ B
S → a|aA|Aa|b
A → aBB
B → a|Aa|b

Notiamo regole terminali come B→Aa, S→Aa, ecc. Si nota che "a" è simbolo terminale, dunque va sostituito con una regola apposita nelle regole che lo compongono
S → a|XA|AX|b
A → XBB
B → AX|b|a
X → a

Similmente ora avremo A → XBB, quindi basterà rimpiazzare XB con una regola, che chiamo Y:
S → a|XA|AX|b
A → CB
B → AX|b|a
X → a
Y→XB

Convertire nella forma normale di Chomsky la seguente grammatica:

S → AbA
A → Aa|ε

Eliminiamo la ε, quindi A → ε
S → AbA|bA|Ab|b
A → Aa|a

Ci sono regole unitarie, ma sono tutte terminali.
Notiamo però che Ab è terminale e va sostituito con una regola apposita.

S →XA|bA|Ab|b
A → Aa|a
X → Ab

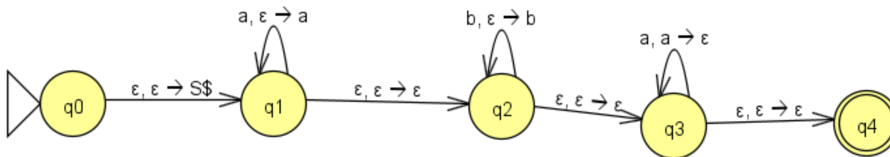
Andiamo quindi a togliere b da bA e a da Aa con regole apposite:

S → XA|YA|AZ|b
A → AY|a
X → Ab
Y → a
Z → b

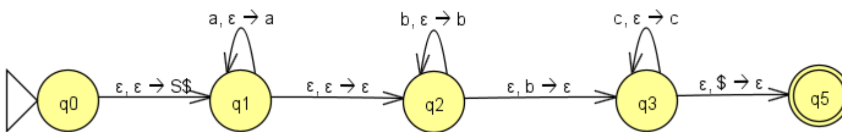
1. Construct pushdown automata for the following languages. Acceptance either by empty stack or by final state.

   (a) $\{ a^n b^m a^n \mid m, n \in \mathbb{N} \}$
   (b) $\{ a^n b^m c^m \mid m, n \in \mathbb{N} \}$

a)

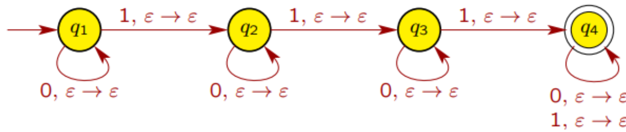


b)



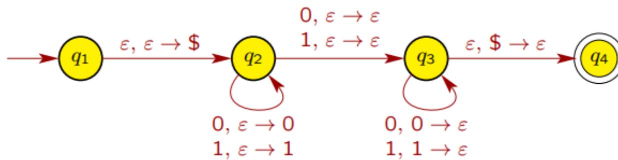
(a) $A = \{ w \in \{0,1\}^* \mid w$ contains at least three 1s $\}$

Answer:



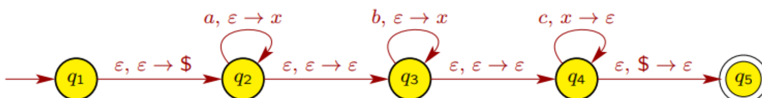
(b) $B = \{ w \in \{0,1\}^* \mid w = w^{\mathcal{R}}$ and the length of w is odd $\}$

Answer:



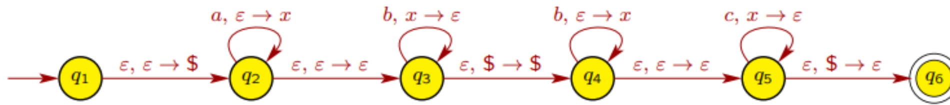
(e) $E = \{ a^i b^j c^k \mid i, j, k \geq 0$ and $i + j = k \}$

Answer:



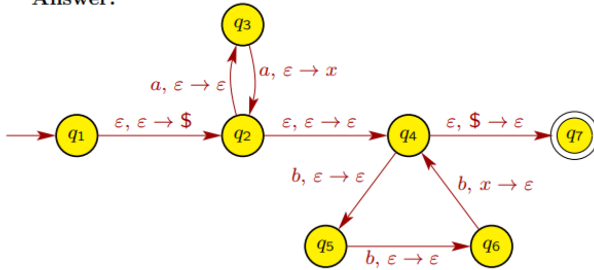
(g) $L = \{ a^i b^j c^k \mid i, j, k \geq 0$ and $i + k = j \}$

Answer: A PDA M for L is as follows:



(f) $F = \{ a^{2n}b^{3n} \mid n \geq 0 \}$

Answer:

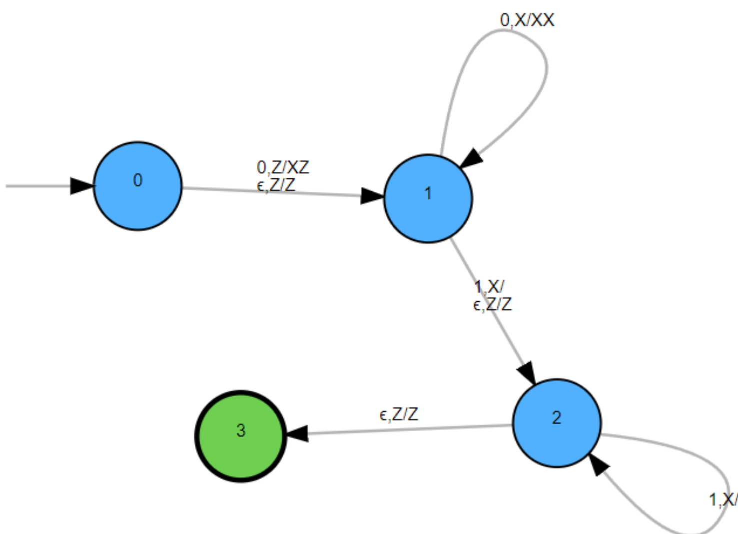


Construct a PDA that recognizes the following language:

{0^n 1^n | n >= 0}

Hint: To get the maximum number of points, use as few states and nonterminals as possible!

- **Alphabet:** {0, 1}
- **Stack alphabet (the first symbol is the initial one):** Z X apply
- **Acceptance condition:** final state
- **Deterministic (DPDA):** false



Esercizi Pumping lemma per linguaggi context-free

- Il linguaggio $L_1 = \{a^n b^n \mid n \geq 0\}$
- Il linguaggio $L_2 = \{a^n b^n c^n \mid n \geq 0\}$
- Il linguaggio $L_3 = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$
- Il linguaggio $L_4 = \{ww^R \mid w \in \{0,1\}^*\}$
- Il linguaggio $L_5 = \{ww \mid w \in \{0,1\}^*\}$

1) Assumiamo che il linguaggio L sia context-free per assurdo.
Pertanto L dovrà avere una pumping length, che chiamiamo "p"
Le condizioni da rispettare sono:

24) uv^ixy^iz vero per ogni i >= 0
25) |vy| > 0
26) |vxy| <= p

Prendiamo poi una stringa $w=a^pb^p$
La dividiamo in 5 parti tali da avere uvxyz
Prendiamo p=3 e avremo quindi:
aaabbb

27) u = a
28) v = aa
29) x = b
30) y = b
31) z = b

Pompiamo con i=3 su uv^2xy^2z ottenendo
aaaaabbbb
Si nota quindi che il numero di a è diverso da quello delle b.
Quindi il linguaggio non è CF.

2) Assumiamo che il linguaggio L sia context-free per assurdo.
Pertanto, L dovrà avere una pumping length, che chiamiamo "p"
Le condizioni da rispettare sono:

32) uv^ixy^iz vero per ogni i >= 0
33) |vy| > 0
34) |vxy| <= p

Prendiamo poi una stringa $w=a^pb^pc^p$
La dividiamo in 5 parti tali da avere uvxyz
Prendendo ad esempio p=5 avremo:
aaaaabbbbbccccc
```
-- --- ------    -- --
u  v    x         y  z
```

A queste condizioni applicando ad esempio uv^2xy^2z
aaaaaaaabbbbbccccccc
Come si può vedere a seguito del pumping, il numero di a, b e c risulta essere sbilanciato. Dunque
Dunque, il linguaggio non è CF.

3) Assumiamo che il linguaggio L sia context-free per assurdo.
Pertanto, L dovrà avere una pumping length, che chiamiamo "p"
Le condizioni da rispettare sono:

1) uv^ixy^iz vero per ogni i >= 0
2) |vy| > 0

3) |vxy| <= p

Prendiamo poi una stringa $w=a^qb^rc^s$
Assumendo per l'appunto tutti gli esponenti >= 0, prendiamo
una stringa w costituita come:
aaabbbbccccc q=3; r=4; s=5

1) u = a
2) v = aa
3) x = bb
4) y = bb
5) z = ccccc

Pompiamo tipo con p=2, avendo quindi uv^2xy^2z e:
aaaaabbbbbbccccc
Il numero di b non è <= k e quindi il linguaggio non è CF.

4) Assumiamo che il linguaggio L sia context-free per assurdo.
Pertanto, L dovrà avere una pumping length, che chiamiamo "p"
Le condizioni da rispettare sono:

1) uv^ixy^iz vero per ogni i >= 0
2) |vy| > 0
3) |vxy| <= p

Prendiamo poi una stringa $w=0^p1^k0^p$
Assumendo per l'appunto tutti gli esponenti >= 0, prendiamo
una stringa w costituita come:
0000011100000 q=5; r=3;

1) u = 00
2) v = 000
3) x = 1110
4) y = 00
5) z = 00

Prendiamo un pumping del tipo uv^2xy^2z avendo ad esempio:
000000001110000000
Si vede quindi che il numero di 0 non è pareggiato da entrambe le parti come invece dovrebbe essendo stringa palindroma, dunque il linguaggio non è CF.

5) Assumiamo che il linguaggio L sia context-free per assurdo.
Pertanto, L dovrà avere una pumping length, che chiamiamo "p"
Le condizioni da rispettare sono:

1) uv^ixy^iz vero per ogni i >= 0
2) |vy| > 0
3) |vxy| <= p

Prendiamo poi una stringa $w=0^p1^k$
Assumendo per l'appunto tutti gli esponenti >= 0, prendiamo
una stringa w costituita come:
0000111111 p=4; k=6;

4) u = 00
5) v = 00
6) x = 111
7) y = 11
8) z = 00

Come pumping prendiamo uv^3xy^3z, avendo quindi:
000000001111111100
Come si vede il linguaggio ha un numero di 0 diverso dal numero di 1 e dunque il linguaggio non può essere CF.

Scritto da Gabriel

11. Usa i linguaggi $A = \{a^m b^n c^n \mid m, n \geq 0\}$ e $B = \{a^n b^n c^m \mid m, n \geq 0\}$ per mostrare che la classe dei linguaggi context-free non è chiusa per intersezione.

L'operazione di intersezione è definita liberamente per i linguaggi regolari; significa quindi che se eseguiamo l'intersezione di due linguaggi context-free, in questo caso A e B, anch'essa deve essere CF. Considerando ad esempio due generiche parole del linguaggio $w=a^p b^q c^p$ e $w=a^q b^p c^q$

9) caso base, con p=q=0 sono: abc & abc, dunque l'intersezione è fattibile
10) caso induttivo, con generici esponenti p, q entrambi > 0 possiamo avere:
    (es. p=2, q=3)
    $a^2b^3c^2$ & $a^3b^2c^3$
    avremo che l'intersezione genererà numero diverso di a, di b, e di c.
    Quindi ad esempio potremmo avere stringhe del tipo:
    $a^2b^5c^3$, $a^5b^3c^3$, ecc.

Eseguendo l'intersezione si nota che otterremmo stringhe diverse e, se volessimo dimostrarlo con il PL per linguaggi CF, otterremmo sempre stringhe sbilanciate. Dunque, l'operazione di intersezione non può essere chiusa per il linguaggi CF.

12. Dimostrare che i seguenti linguaggi sono context-free. Salvo quando specificato diversamente, l'alfabeto è $\Sigma = \{0, 1\}$.

(a) $\{w \mid w$ contiene almeno tre simboli uguali a 1$\}$
(b) $\{w \mid$ la lunghezza di w è dispari$\}$
(c) $\{w \mid w$ inizia e termina con lo stesso simbolo $\}$
(d) $\{w \mid$ la lunghezza di w è dispari e il suo simbolo centrale è 0$\}$
(e) $\{w \mid w = w^R$, cioè w è palindroma$\}$
(f) $\{w \mid w$ contiene un numero maggiore di 0 che di 1$\}$
(g) Il complemento di $\{0^n 1^n \mid n \geq 0\}$
(h) Sull'alfabeto $\Sigma = \{0, 1, \#\}$, $\{w\#x \mid w^R$ è una sottostringa di x e $w, x \in \{0, 1\}^*\}$
(i) $\{x\#y \mid x, y \in \{0, 1\}^*$ e $x \neq y\}$
(j) $\{xy \mid x, y \in \{0, 1\}^*$ e $|x| = |y|$ ma $x \neq y\}$
(k) $\{a^i b^j \mid i \neq j$ e $2i \neq j\}$

Per dimostrare che i linguaggi *sono* context-free dobbiamo necessariamente fare delle derivazioni.
Se dovessimo dimostrare che *non* sono context-free allora dovremmo usare il PL per CFL.

a) S → 1X1X1X1X
   X → 0X|1X|ε
b) S → 0A|1A
   A → 0S|1S|ε
c) S → 0A|1B
   A → 0|ε
   B → 1|ε
d) S → 0S0|1S1|0|0S1|1S0
e) S → 0|1|1S1|0S0|ε
f) S → T0T
   T → 1T0|0T1|0T0|ε
g) S → 1A|0A
   A → A0|
h) S → A
   A → #B|0A0|1A1
   B → 0B|1B|ε

i) S → A#B|B#A|ε
A → TAT|0
B → TBT|1
T → 0|1

j) S → AB|BA
A → 0|0A0|0A1|1A0|1A1
B → 1|0B0|0B1|1B0|1B1

k) S → S_1|S_2
S_1 → aA
A → bAb|aA|ε
S_2 → Bb!aBb
B → Bb|aaBb|aBb|ε

13. Se A e B sono linguaggi, definiamo $A \circ B = \{xy \mid x \in A, y \in B \text{ e } |x| = |y|\}$. Mostrare che se A e B sono linguaggi regolari, allora $A \circ B$ è un linguaggio context-free.

Se A e B sono linguaggi regolari, allora sono descrivibili mediante le medesime operazioni dei linguaggi regolari e successivamente almeno da un automa DFA/NFA.
Sia M=(Q, Σ ,δ, q_o, F) un DFA che riconosce A.
M'=(Q', Σ , δ', q_o, F')
Descriviamo quindi A, composto evidentemente da stringhe "x", poi B, composto da stringhe di tipo "y"
Significa quindi che per questi dettagliamo:
(r_A, L) → (s_A,M) se $\delta_L(r_A, x)=s_A$
(r_B, L) → (s_B,M) se $\delta_L(r_B, y)=s_A$
Similmente è definita (r_A, L, M) → (s_b,M, L) se $\delta_L(r_A, x)=\delta_L(r_B, y)$.
A queste condizioni notiamo che il linguaggio context-free necessariamente richiede che gli stati finali corrispondano tra i due linguaggi.

Articoliamo quindi come idea di stato finale F (δ_0, x), F(δ_1, y) = F(X • Y)
Necessariamente l'idea è che, nel caso non esista uno dei due stati,
lo stato finale deve essere:

F(X • Y) = F (δ_0, x)
oppure

F(X • Y) = F(δ_1, y)

per la caratterizzazione di termini di cui sopra. Dunque, il linguaggio è CF.

Tutorato 5

A = {0^R | r = 2^k, k in N}
PL
Per ogni w → w = xyz , |y| > 0, |xy| <= k

Per ogni i >= 0, xy^iz per ogni L

Quindi per ognuno esiste un i tale che xy^iz non appartiene ad L.

Esempio: {0^n1^n, n > 0}
0^k1^k ,
Obiettivo: Esiste i > 0. i ≠ 0, y = 0^a a > 0

i=0 $0^{k-z}1^k$ non appartiene ad L

-k
-$w=0^{2^k}$ $n=2^k$ $2^k > k$
-$y=0^p$ $p<k$ $2^{10} > 10$
-$x=\varepsilon$
-z = tutto il resto

i=0 $xy^iz = 0^{(2^k - P)}$

Devo dimostrare che il valore non è una potenza di 2.
- $2^6 - 32 = 64$
| |
2^K p

- 2^6 - 17

L'idea è di parametrizzare tutta la dimostrazione basata sul singolo valore, tale che sia funzionale al valore di k ignoto, in maniera tale che la parola pompata non appartenga al linguaggio sfruttando il fatto sia un generico "k".
$2^2, 2^3, 2^4 \ldots . 2^K, 2^{K+1}$
- 2^4, 2^5, $2^{4+1} = 2^4 * 2 = 2^4 + 2^4$
16 32 = 16 + 16

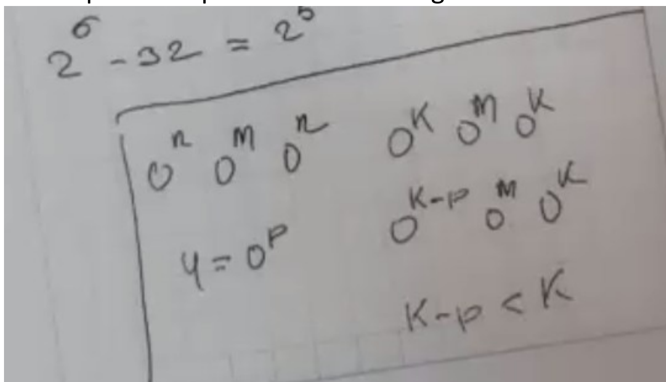
$2^{K+1} = 2^K + 2^K$
Diciamo quindi che al passo k-esimo ci sia almeno un 2k come distanza, in seguito soprattutto al pumping.
Usando un "i" positivo, produco una parola di lunghezza $2^K + x$, tale che $x < 2^K$

$y=0^P$ $x=\varepsilon$ $z=0^{(2^K - p)}$

Questo porta a dire che $w = 0^P0^{2^k-P}$

i=2 $xy^2k = (0^P)^2\ 0^{(2^K-p)}$ = $0^{2p}0^{2^K-p} = 0^{2^K - p}$

Dunque diciamo che p < k, in quanto per il PL deve essere |xy| <= k e quindi p < 2.
Usciamo dal linguaggio in quanto non otteniamo esattamente l'esponente 2^K e quindi come al solito si ha la stringa sbilanciata.
La dimostrazione è a prova di "k", sappiamo solo che esiste ma non facciamo nulla con quel valore.
Non si può dire quindi una roba del genere:



Cioè che vale sempre (ad esempio il numero primo, tale che non tutti gli esponenti risolvono il discorso del PL e si richiede un ragionamento tramite dimostrazione).

L={0^m1^n | n/m è un numero intero} (ragionando sul fatto che il numero possa non essere intero)
Importante che quando si applica il PL non si applichi automaticamente.

- A, B shuffle perfetto
{w | w = $a_1b_\backslash$ .....a_kb_k con a_1, ... a_k ∈ A, b_1, ... b_k ∈ B a_i, b_i ∈ ∑}

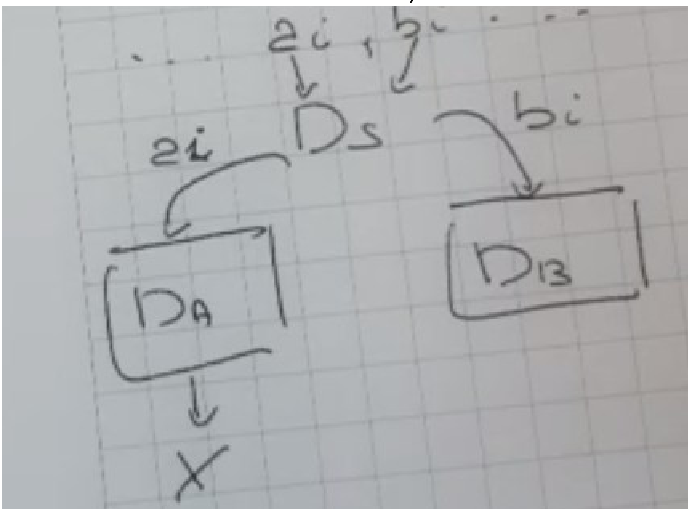
A,B sono regolari → Shuffle è regolare
a=|1|0|1|
b=|0|0|1|
w=|10|00|11

∑* = (0+1)* indica tutte le stringhe (compresa la vuota) ottenibili da linguaggio

Sappiamo quindi che esistono due automi del tipo:
D_A = {Q_A, ∑, δ_A, q_A, F_A}
D_B = {Q_B, ∑, δ_B, q_B, F_B}

Vogliamo produrre quindi:
D_S = {Q_S, ∑, δ_S, q_S, F_S}

L'automa quindi sa l'ordine e che gli elementi sono invertiti, conseguentemente si va comunque almeno ad uno stato accettante.
L'idea informale è di costruire un automa che prende la posizione generica, richiama la procedura e poi non vedendo il funzionamento interno, richiamo i simboli avendo una dimostrazione generica:



<u>Dim. formale</u>

D_S = {Q_S, ∑, δ_S, q_S, F_S}
Generica $\delta(q_X, a)$ → q

Ad esempio metto come input il simbolo "a":
δ((x,y,A), a) → ($\delta_{A(X,a)}$,y, B)

L'idea quindi è che gli input si alternano e mi resta da aggiornare lo stato in cui si trova A, usando le sue funzioni di transizione.

x stato corrente D_A
y stato corrente D_B
A flag

$\delta((x, y, B), b) \rightarrow (x, \delta_{B(y,b)}, A)$

$\delta(q,x) \rightarrow q$
L'idea è che la tupla rappresenti gli stati dell'automa, ottenendo gli stati nuovamente e riorganizzandoli, costruendo automaticamente la funzione di transizione di δ_S.

Quindi l'idea è di avere un aggiornamento degli stati tali da avere le transizioni ricombinate, rimanendo con gli input uguali

- Q_S

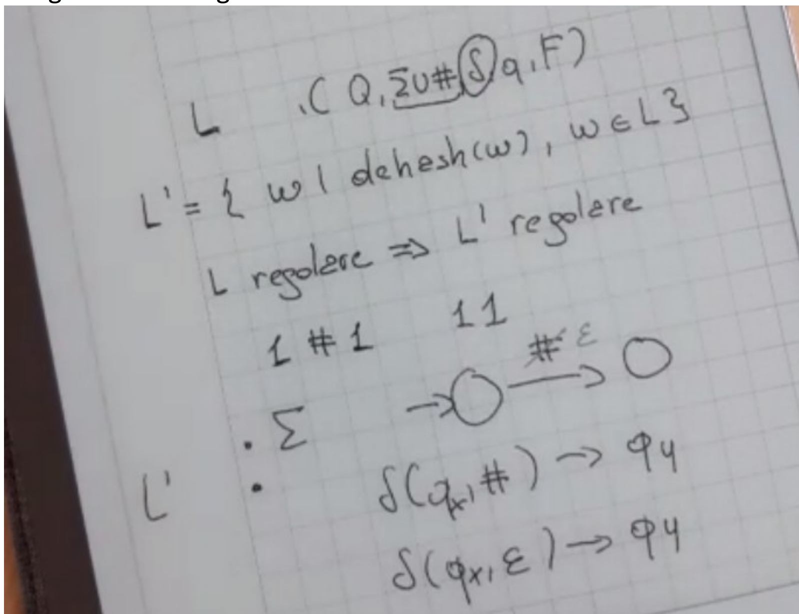
$Q_A \times Q_B \times \{A, B\}$
Usiamo quindi "x" come prodotto cartesiano, tipo avendo
$\{a,b\} \times \{c,d\}$ eseguo il prodotto tra insiemi (prodotto cartesiano)
$=\{(a,c), (a,d), (a,c), (b,d), (b,c)\}$

$\{q_x, q_y\} \times \{q_Z\} \times \{A, B\}$
$\{(q_X, q_Z), (q_Y, q_Z)\} \times \{A, B\}$
(q_X, q_Z, A).....

La risposta è: creare gli stati come prodotto (producendo tutto "brutalmente" con tutte le possibili combinazioni).
Stato iniziale q_S
$q_S = (q_A,q_B,A)$
$\delta[(q_A,q_B,A), a_{1]} \rightarrow (\delta(q_A, q_1))$

$F_A \times F_B \times$
b_K
$\delta((q_X, q_y, B), b_K)$

$L'=\{w \mid dehash(w), w \in L\}$
L regolare $\rightarrow$ L' regolare



L linguaggio regolare su Σ
$L' = \{y \mid xy \in L \text{ per } x \in \Sigma^*\}$
$w=1011$, $w \in L$

1
11
011
1011
ε

(Q, δ, q, F, ∑)

Ex 3

Sia L un linguaggio regolare su un alfabeto Σ con # ∈ Σ e sia dehash(w) la funzione che rimuove il simbolo hash dalla stringa. Ad esempio dehash(1#1) = 11, dehash(0#10#) = 010. Dimostrare che il linguaggio dehash(L) = $\{dehash(w) : w \in L\}$ e' regolare.

Partiamo con la considerazione del fatto che gli automi condividono lo stesso alfabeto ∑.
Diamo quindi due linguaggi regolari A e B formati come:

$D_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$
$D_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$

Consideriamo la funzione di transizione formata dai simboli:

$\delta((q_0, ,\#0\#),1) = \delta(\delta(q_0, ,0),1)$
$\delta((q_0, ,\#1\#),1) = \delta(\delta(q_0, ,1),1)$
$\delta((q_0, ,\#1\#),0) = \delta(\delta(q_0, ,0),1)$
$\delta((q_0, ,\#1\#),0) = \delta(\delta(q_0, ,1),0)$

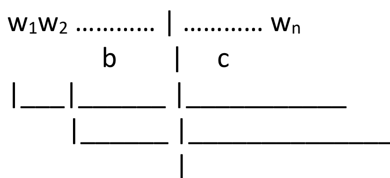
Consideriamo quindi che tutti i simboli di partenza vadano allo stato successivo perdendo le stringhe hash.
Per completare l'automa definiamo l'insieme degli stati finali, che sarà dato dal prodotto di tutti gli stati raggiunti senza le stringhe hash
$Q_S = Q_A \times Q_B \times \{(A, B)/\#\}$
definendo quindi la mancanza dello hash.
Lo stato iniziale sarò sempre definito da $q=(q_A, q_B, A)$ e lo stato accettante è dato dai simboli $F = F_A \times F_B \times \{A]$, dunque l'input era quello precedemente letto da B.

Ex 4

Sia L un linguaggio regolare su un alfabeto Σ. Dimostrare che il linguaggio suffixes(L) = $\{y|xy \in L$per qualche stringa x $\in \Sigma^*\}$ e' regolare.

Sia G la grammatica che genera L.
Assumiamo che G sia in forma normale di Chomsky.
Avremo quindi regole del tipo:
A → BC
D → d

Siccome c'è una concatenazione, un pezzo della parola sarà generato da "b" e un altro da "C"



A' suffissi delle parole generate da A
A' → A|B'C|C|C'|ε

Quindi quello che viene adesso è la variabile iniziale

Scritto da Gabriel

D’ → d|ε
S’ è la variabile iniziale sse S è variabile iniziale di G

Ex 5

Esercizi aggiuntivi

1. Dimostrare che il linguaggio $\{0^m 1^n | n/m$ e’ un numero intero$\}$ non e’ regolare

Usiamo il Pumping Lemma per dimostrare che il linguaggio non è regolare.

Supponiamo per assurdo che L sia regolare:

- sia k la lunghezza data dal Pumping Lemma;
- consideriamo la parola $w = 0^{k+1}1^{k+1}$, che è di lunghezza maggiore di k ed appartiene ad L perché $(k+1)/(k+1) = 1$;
- sia $w = xyz$ una suddivisione di w tale che $y \neq \varepsilon$ e $|xy| \leq k$;
- poiché $|xy| \leq k$, allora x e y sono entrambe contenute nella sequenza di 0. Inoltre, siccome $y \neq \varepsilon$, abbiamo che $x = 0^q$ e $y = 0^p$ per qualche $q \geq 0$ e $p > 0$. z contiene la parte rimanente della stringa: $z = 0^{k+1-q-p}1^{k+1}$. Consideriamo l’esponente $i = 0$: la parola xy^0z ha la forma

$$xy^0z = xz = 0^q 0^{k+1-q-p}1^{k+1} = 0^{k+1-p}1^{k+1}.$$

Si può notare che $(k+1-p)/(k+1)$ è un numero strettamente compreso tra 0 e 1, e quindi non può essere un numero intero. Di conseguenza, la parola non appartiene al linguaggio L, in contraddizione con l’enunciato del Pumping Lemma.

2. Siano L e M due linguaggi regolari su alfabeto {0,1}. Dimostare che il linguaggio L&M $= \{x\&y | x \in L, y \in M, |x| = |y|\}$, dove x&y e’ l’and logic bit a bit. Per esempio, 101&001 = 001.

Partiamo con la considerazione del fatto che gli automi condividono lo stesso alfabeto ∑-
Diamo quindi due linguaggi regolari A e B formati come:
$D_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$
$D_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$
Consideriamo la funzione di transizione formata dai simboli:
$\delta((x, y, A), a) = (\delta_A(xy), B))$ se x=y
$\delta((x, y, A), a) = (\delta_A(x), B)) \vee (\delta_A(y), B))$ se x≠y
Essendo AND logico bit a bit dei due linguaggi precedenti, con la notazione usata ragioniamo sul fatto che dal simbolo iniziale andremo al successivo B avendo l’effettivo AND logico dei bit presentati. Segue l’insieme degli stati definito come:
$Q_S = Q_A \times Q_B \times \{(A, B)\}$
Essendo prodotto cartesiano in particolare, sappiamo già da questo che gli effettivi stati ottenibili, per proprietà commutativa, saranno i bit definibili come prodotto l’uno dell’altro.
Lo stato iniziale sarò sempre definito da $q=(q_A, q_B, A)$ e lo stato accettante è composto da $F_{AB}=F_A \times F_B$

3. Siano L e M due linguaggi regolari su alfabeto {0,1}. Dimostare che il linguaggio $LXORM = \{xXORy | x \in L, y \in M, |x| = |y|\}$, dove $xXORy$ e’ l’and logic bit a bit. Per esempio,$101XOR001 = 100$.

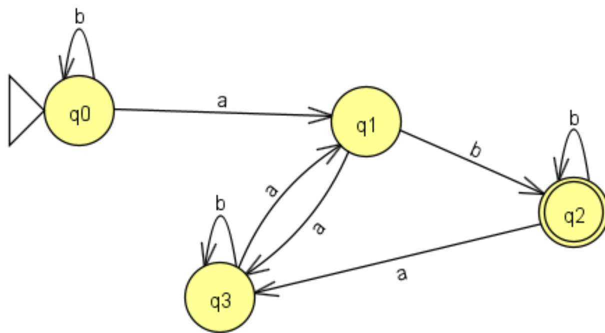
Partiamo con la considerazione del fatto che gli automi condividono lo stesso alfabeto ∑-
Diamo quindi due linguaggi regolari A e B formati come:
$D_A = (Q_A, \Sigma, \delta_A, q_A, F_A)$
$D_B = (Q_B, \Sigma, \delta_B, q_B, F_B)$
Consideriamo la funzione di transizione formata dai simboli:
$\delta((x, y, A), a) = (\delta_A(xy), B))$ se x≠y
$\delta((x, x, A), a) = (\delta_A(x), B))$
oppure se x=y

$\delta((y, y, A), a) = (\delta_A(y), B))$
Essendo uno XOR è operazione disgiuntiva come operazione e dunque avremo “1 se uguali, 0 se sono diversi”, almeno idealmente provando a rappresentarlo in maniera concreta.
$Q_S = Q_A \times Q_B \times \{(A, B)\}$
con tutte le combinazioni di A e B
con stato iniziale $q=\{q_A, q_B, A)$
e stato accettante definibile come $F=F_A \times F_B \times \{A\}$, definendo per continuità lo stato finale con i precedenti.

Esercizi vari (appelli/compitini)

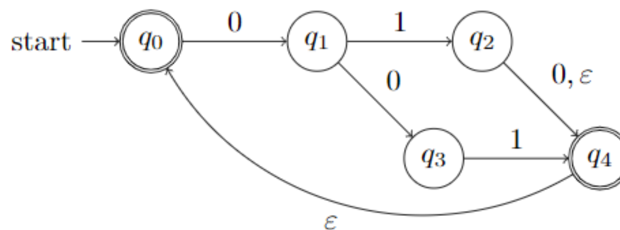
Primo compitino del 12 Aprile 2019

Considerare il linguaggio $L = \{w \mid w \in \{a, b\}^*$ con un numero dispari di a e che terminano con $b\}$



Dare un'espressione regolare che rappresenti il linguaggio L.

Dato il seguente NFA

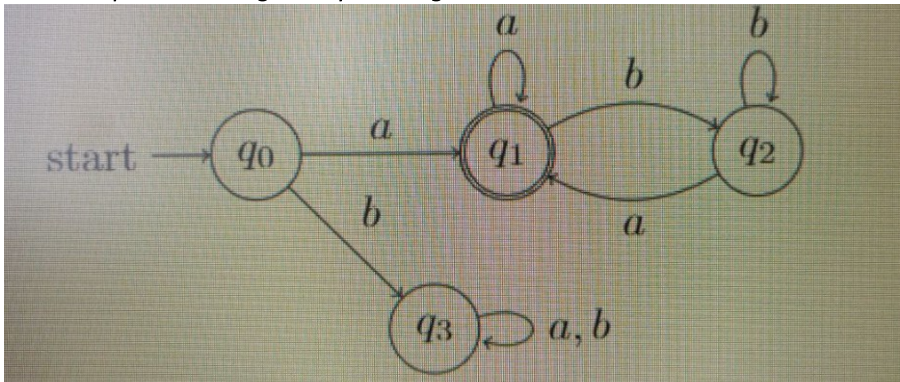


Si trova la ε-chiusura, che sarebbe q0,q2,q4.
Utile perché come si vede anche dall’automa, ogni stato che comprende singolarmente q1, q2, q4 dovrà per forza comprendere tutta la chiusura, quindi (q0,q2,q4).

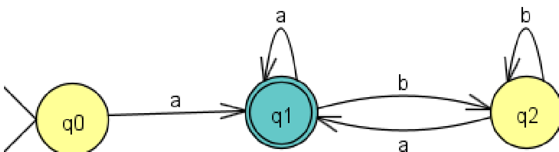
	0	1
->q0	q1	Ø
q1	q3	q2
q2	q4	Ø
q3	Ø	(q0,q4)
q4	q1	Ø
*(q0,q4)	q1	Ø
*(q0,q2,q4)	(q0,q1,q4)	Ø
*(q0,q1,q4)	(q1,q3)	(q0,q2,q4)
(q1,q3)	q3	(q0,q2,q4)

Completare lo schema con il Gioco del Pumping Lemma per far vincere il giocatore 2 (quindi si dimostra che non è regolare).
Giocatore 1: Sceglie 4 come valore di h
Giocatore 2: Sceglie la parola w = bbbb=bbbb
Giocatore 1: Suddivide la stringa xyz in:
x=b,   y=bb   z=b=bbbb
Giocatore 2: Sceglie un i tale che xy^iz, per esempio con i=3
ed avremo bbbbbbbb=bbbb
Vedendo che la stringa non è uguale da entrambe le parti *l'espressione non può essere considerata regolare.*

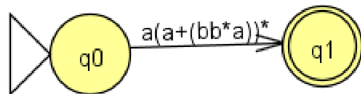
Dai un'espressione regolare per il seguente automa:



Elimino direttamente q3, non porta a nessuno stato accettante.



Successivamente elimino q2, essendo stato non finale.
Si nota che q1 ha due stati uscenti e sarà unione di a con tutto il ciclo (bb*a), unito con l'a esterno e tutto ripetuto.
Conseguentemente si ha: a(a+bb*a)* e unito con l'a fuori da tutto.



ER finale: a(a+(bb*a))*

<u>Dal file "esercizi-preparazione-esame"</u>

Dati i linguaggi A e B, lo *shuffle perfetto* di A e B è il linguaggio

$$\{w \mid w = a_1b_1a_2b_2 \dots a_kb_k, \text{ dove } a_1a_2 \dots a_k \in A \text{ e } b_1b_2 \dots b_k \in B, \text{ ogni } a_i, b_i \in \Sigma\}$$

Mostrare che la classe dei linguaggi regolari è chiusa rispetto allo shuffle perfetto, cioè che se A e B sono linguaggi regolari allora anche il loro shuffle perfetto è un linguaggio regolare.

La classe dei linguaggi regolari è chiusa rispetto al linguaggio dello shuffle perfetto in quanto il linguaggio presentato è fondamentalmente un'elaborata concatenazione, operazione chiusa per sua stessa definizione. Ipotizzando infatti nel caso base un linguaggio formato da due stati iniziali (a_1b_1) notiamo subito che si può banalmente rappresentare per entrambi.
Induttivamente quindi si può definire per qualsiasi quantità compresa da 1 a k, in quanto sappiamo essere valida la proprietà di chiusura di Kleene, pertanto valida anche in caso di esponente k-esimo.

Alla luce di queste considerazioni, possiamo affermare il linguaggio sia regolare in quanto rappresentato da un automa a stati finiti. Infatti le operazioni sui singoli pezzi sono regolari. Su a_1b_1 e le singole parti, infatti, la presenza alternata di stati regolari contribuisce ad essere tutta concatenazione regolare.
Anche nel caso di una ipotetica applicazione del Pumping Lemma, si vede subito che in effetti qualsiasi esponente k non cambia quali siano le stringhe rappresentate, pur "pompando" la possibile parola.
Dunque il linguaggio è chiaramente regolare.

Sia $\Sigma = \{0, 1\}$, e considerate il linguaggio

$$D = \{w \mid w \text{ contiene un ugual numero di occorrenze di 01 e di 10}\}$$

Mostrare che D è un linguaggio regolare.

Ipotizzando il linguaggio sia regolare, valenti dunque le classiche condizioni:
y≠ε, w=xyz, |xy|<=z
considerando ad esempio:
x=ε y=(01)* z=(10)*
l'unica possibile suddivisione che soddisfi la consegna è questa, conseguentemente si vede che il linguaggio rimane regolare, avendo contemporaneamente la concatenazione dei due linguaggi comn l'inserimento forzato delle due stringhe.

Il pumping lemma afferma che ogni linguaggio regolare ha una lunghezza del pumping p, tale che ogni stringa del linguaggio può essere iterata se ha lunghezza maggiore o uguale a p. La *lunghezza minima del pumping* per un linguaggio A è il più piccolo p che è una lunghezza del pumping per A. Per ognuno dei seguenti linguaggi, dare la lunghezza minima del pumping e giustificare la risposta.

(a) 110^*
(b) $1^*0^*1^*$
(c) $0^*1^*0^*1^* + 10^*1$
(d) $(01)^*$
(e) $\emptyset$
(f) $0^*01^*01^*$
(g) $10(11^*0)^*0$
(h) 101101
(i) $\{w \in \Sigma^* \mid w \neq 101101\}$
(j) 0001^*
(k) 0^*1^*
(l) $001 + 0^*1^*$
(m) ε
(n) $1^*01^*01^*$
(o) 1011
(p) Σ^*

(a) 3: si necessita di avere almeno tre caratteri per poter cominciare ad eseguire il pumping. Infatti parole come 00, 11 o similari non sono acettate.
(b) 3: parole come 11 (cioè xy^0z) non sono accettate per costruzione, prendendo ad esempio 101 come xyz si nota che devono esserci entrambi per poter cominciare a pompare.
(c) 3: L'unione c'è ma non ci interessa, la proprietà vale comunque. Inoltre si nota che devo avere almeno due 1 ma l'idea è la stessa dell'esercizio precedente.
(d) 1: l'insieme vuoto deve avere almeno una stringa con cui operare e quindi pompare all'infinito una parola.
(e) 2: abbiamo bisogno di entrambi i caratteri per pompare.
(f) 3: le parole devono essere divise e definite in 3 pezzi anche qui per formare una parola valida e pompare.
(g) 4: in pratica la lunghezza deve essere 4 in quanto potrebbe esserci una suddivisione che non sbilancia la stringa avendo soli tre caratteri, avendo stringhe che pompate non sarebbero nel

linguaggio. Sapendo che (11*0)* è l'espressione minima, avremo quantomeno bisogno di 0 oppure 1 per cominciare a pompare.

(h) 4: si vede infatti che ipotizzando 3 come lunghezza minima, potremmo avere una stringa del tipo 111 che rimane sempre regolare.

(i) 3: se sappiamo (penso io) che la stringa precedente non fa parte del linguaggio, significa considerando l'alfabeto precedente che necessitiamo di almeno tre caratteri per poter avere una stringa pompabile, considerando il caso complementare a quello descritto sopra.

(j) 4: potremmo banalmente avere la stringa 000 che non sarebbe pompabile.

(k) 2: anche qui, scegliendo 1 non sarebbe pompabile; con 2 almeno avremmo il possibile caso 01, regolarmente pompabile.

(l) 3: lasciando stare il caso dell'unione, comunque si nota che la stringa 001 non può avere 2 come lunghezza minima pompabile.

(m) 1: la stringa vuota deve necessitare di almeno un carattere qualsiasi.

(n) 3: si considera infatti che la minima stringa effettivamente pompabile sia 001

(o) 5: di fatto 1011 potrebbe non essere accettata dal linguaggio come pompata, perché già integralmente parte del linguaggio stesso.

(p) 2: considerando che la stringa vuota necessita di almeno un carattere e l'alfabeto la comprende di sicuro essendo star, potrebbe bastare per un generico alfabeto avere un solo altro carattere.

MPL (Minimum Pumping Length) Extra:
aab U a*b* → 1: siccome la parte di sinistra può essere ugualmente generata dalla parte di destra, la lunghezza minima deve essere per forza 1.

__Prime parti dei due appelli 2021:__

Dimostra che se L ed M sono linguaggi regolari sull'alfabeto $\{0,1\}$, allora anche il seguente linguaggio è regolare:

$$L \sqcap M = \{x \sqcap y \mid x \in L, y \in M \text{ e } |x| = |y|\},$$

*dove $x \sqcap y$ rappresenta l'*and bit a bit *di x e y. Per esempio, $0011 \sqcap 0101 = 0001$.*

Essendo regolari entrambi i linguaggi, posso costruire un automa con l'unione chiusa del linguaggio che ha come stato iniziale q_i, come insieme di stati tutte le coppie tutte le coppie di stati finali.
L'insieme delle transizioni:

$$\delta((r_L, r_M), 0) = \{(\delta_L(r_L, 0), \delta_M(r_M, 0)), (\delta_L(r_L, 1), \delta_M(r_M, 0)), (\delta_L(r_L, 0), \delta_M(r_M, 1))\}$$
$$\delta((r_L, r_M), 1) = \{(\delta_L(r_L, 1), \delta_M(r_M, 1))\}$$

Tanto ci basta per confermarne intuitivamente la regolarità.

Considera il linguaggio

$$L_2 = \{w \in \{0,1\}^* \mid w \text{ contiene lo stesso numero di } 00 \text{ e di } 11\}.$$

Dimostra che L_2 non è regolare.

Consideriamo come sempre k la lunghezza del PL, considerando come sempre una parola tale che:
$y \neq \varepsilon$ e $|xy| \leq k$, una parola $w=0^k1^k$.
Consideriamo quindi due esponenti p, q entrambi maggiori di 0 per cui avremo:
$x=0^p$ $y=0^q$ z conterrà sempre la parte rimanente della stringa, espressa come $0^{k-q-p}1^k$
Pertanto elevando a xy^0z avremo $0^p0^{k-q-p}1^k$ avendo poi $0^{k-p}1^k$, come sempre sbilanciata.

Dimostra che se L è un linguaggio context-free, allora anche L^R è un linguaggio context-free.

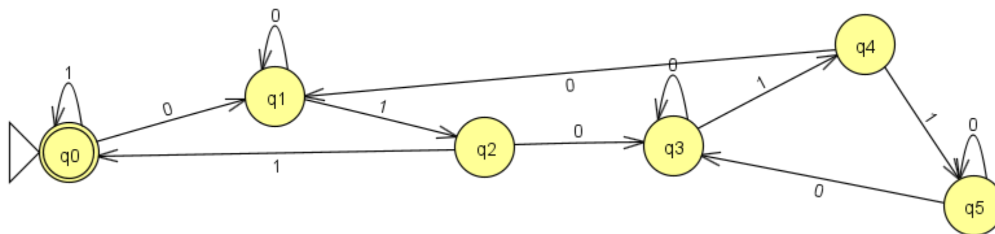
Come ormai sappiamo, un linguaggio L è context-free sse esiste una grammatica G che lo genera tale che essa sia in forma normale di Chomsky.
Quindi l'idea della grammatica è:
A → BC (questi simboli non terminali)
A → b (simbolo terminale)

Nel caso di L^R avremo:
A → CB (in maniera contrapposta)
A → b (simbolo terminale)

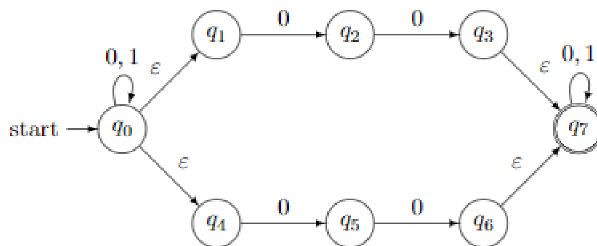
<u>*Prime parti Appelli 1-2 del 2019*</u>

1. Definire un automa a stati finiti (di qualsiasi tipologia) che riconosca il linguaggio

$$L = \{w \in \{0,1\}^* \mid w \text{ contiene un numero pari di occorrenze della sottostringa } 010\}$$



Dato il seguente ε-NFA

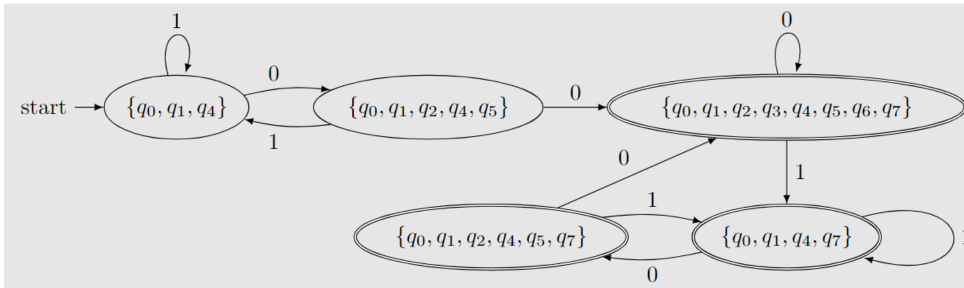


costruire un DFA equivalente. Dare solo la parte del DFA che è raggiungibile dallo stato iniziale.

Partiamo dalle ε-chiusure, per esempio considerando:
q0,q1,q4 e q3,q6,q7
Poi scriviamo la tabella di transizione:

q_i	0	1
->{q0, q1, q4}	{q0,q1,q2,q4,q5}	{q0,q1,q4}
{q0,q1,q2,q4,q5}	{q0,q1,q2,q3,q4,q5,q6,q7}	Ø
{q0,q1,q2,q3,q4,q5,q6,q7}	{q0,q1,q2,q3,q4,q5,q6,q7}	{q0,q1,q4,q7}
{q0,q1,q4,q7}	{q0,q1,q2,q4,q5,q7}	{q0,q1,q4,q7}

da cui poi si ha:



Sia $\Sigma = \{0, 1\}$ e considerate il linguaggio

$$M3N = \{0^m 1^n \mid m \leq 3n\}$$

(a) Completate il seguente schema di partita per il Gioco del Pumping Lemma in modo da far vincere il Giocatore 2:

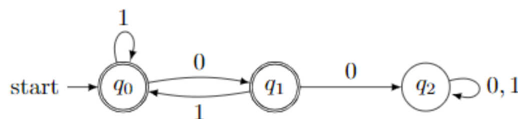
Quando gioca il giocatore 2 significa che il linguaggio *non è regolare*.
Quindi:

11) giocatore 1 sceglie una pumping length h=3
12) giocatore 2 sceglie la parola w=0000111
13) giocatore 1 sceglie una suddivisione della parola
    x=00 y=00 z=111
14) giocatore 2 sceglie una potenza i tale che xy^iz diventi non regolare.
    Per esempio consideriamo y=2 avendo quindi 000000111 che non appartiene al linguaggio e dunque *non può essere regolare*.

(b) Dimostrate che $M3N$ non è un linguaggio regolare usando il Pumping Lemma.

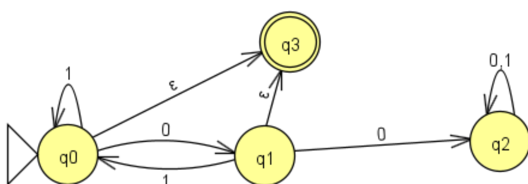
Dimostrare che questo vale significa partire da una parola $w=0^{3h}1^h$, dove h è la pumping length, con le solite: y ≠ ε e |xy| ≤ k.
Per esempio quindi pompando sull'esponente avremo un numero di x e di y composto solo da 0, inoltre considerando anche y non vuota e sempre formata da zeri, qualsiasi pumping porta ad avere una cosa del tipo 0^{3h+k} dove k può essere un qualsiasi esponente pump che sbilancia la suddivisione. Dunque tanto basta per dire che non è regolare.

Considerate il DFA che riconosce il linguaggio di tutte le stringhe sull'alfabeto $\{0, 1\}$ che *non contengono* la sottostringa 00:

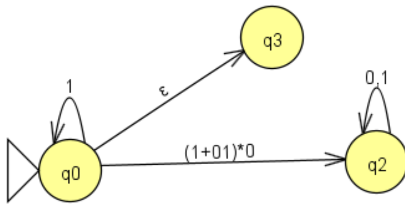


Trasformate il DFA in una Espressione Regolare usando l'algoritmo di eliminazione degli stati.

Prima di tutto come al solito devo trasformare tutti gli stati finali in stati non finali, successivamente aggiungere una ε-transizione al nuovo stato finale:



Procediamo ad esempio eliminando q1, in cui si nota il percorso (1+01)*0 e successivamente:



Si vede quindi che ha due possibili percorsi in uscita, concatenando il precedente percorso con ε da una parte mentre 0 dall'altra, avendo in forma compatta (1+01)*(0+ε).
Questa infatti è l'espressione finale, estesa sarebbe come si vede anche da qui (1+01)* + (1+01)*0.

Usate la soluzione dell'esercizio precedente per creare una Espressione Regolare che definisca il linguaggio di tutte le stringhe sull'alfabeto $\{0, 1\}$ tali che tutte le occorrenze di 11 appaiono prima di tutte le occorrenze di 00.
Partendo dalla precedente significa che comincio a considerare (1+01)* come prima combo; successivamente potrò avere 00 e dunque eseguo la cosa opposta, quindi (0+10)*. Avanzando ancora, potrò avere la stringa vuota ma potrei ancora avere una occorrenza di 0 oppure 1. Abbiamo considerato il caso in cui c'era 0 oppure 1; se non ci sta nessuno dei due, posso avere la stringa vuota e poi avere anche 1.
Quindi posso avere in forma estesa (0+10)*(1+ε) e analogamente (1+01)*(0+ε), quindi (0+10)*(1+ε) (1+01)*(0+ε) e in forma più compatta, considerando che non c'è zero posso avere direttamente uno, la considerazione è avere: (1+01)* (0+10)*(1+ε).

Sia $\Sigma = \{0, 1\}$ e considerate il linguaggio

$$LMN = \{0^\ell 1^m 0^n \mid \ell < n\}$$

(a) Completate il seguente schema di partita per il Gioco del Pumping Lemma in modo da far vincere il Giocatore 2:

Quindi:

15) giocatore 1 sceglie una pumping length h=5
16) giocatore 2 sceglie la parola w=0011100000 che ha lunghezza l > k
17) giocatore 1 sceglie una suddivisione della parola
    x=001 y=110 z=0000
    rispettando le solite condizioni
18) giocatore 2 sceglie una potenza i tale che xy^iz diventi non regolare.
    Per esempio consideriamo y=2 avendo quindi 0011101100000 che non appartiene al linguaggio e dunque non può essere regolare, avendo infatti un numero sbilanciato di 0 e di 1.

Dimostrate che LMN non è un linguaggio regolare usando il Pumping Lemma.
Supponiamo per assurdo la parola sia regolare.
Dando quindi una generica parola con una pumping length "h" e sapendo che y≠ε, w=xyz, |xy|<=k prendiamo ad esempio $w=0^h1^k0^{h+k}$ appartenente al linguaggio (xyz sono qui 0,1,0 singolarmente presi).
Notiamo già che per un generico k 1 permane e gli 1 possono potenzialmente sbilanciarsi.
Quando i=0 nella seconda stringa, si ha un uguale numero di 0 tra x e z e non sarebbe regolare.
Notiamo infatti che normalmente qui, la stringa sarebbe composta da un numero di 0 maggiore del numero presente di 1. Dunque concludiamo che non sia regolare.

Terzo appello 2021 – Prima parte

(8 punti) *Considera il linguaggio*

$$L = \{0^m 1^n \mid m/n \text{ è un numero intero}\}.$$

Dimostra che L non è regolare.

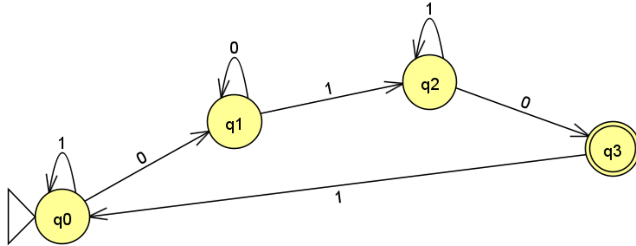
Usiamo il Pumping Lemma per dimostrare che il linguaggio non è regolare.

Supponiamo per assurdo che L sia regolare, pertanto avremo la solita *k* come pumping length e consideriamo come parola $w=0^k1^{k+1}$.
Scegliendo poi come al solito $y \neq \varepsilon$, $|xy| <= k$ e una suddivisione $w=xyz$, sappiamo che 0^m ed 1^n sono sicuramente regolari per qualche esponente $k>0$ e $p>0$. Dunque z conterrà la rimanente parte della stringa, circa $0^{k+1-q-p}1^{k+1}$. Per $i=0$ come pumping, avremo $0^{k+1-p}\,1^{k+1}$. In tali condizioni la stringa si sbilancia ed il linguaggio non è più regolare.

<u>*Conferma voto secondo appello 2021*</u>

Definire un automa a stati finiti (di qualsiasi tipologia) che riconosca il linguaggio

$$L_1 = \{w \in \{0,1\}^* \mid w \text{ contiene un numero pari di occorrenze di } 010\}$$



Definire una grammatica context-free che generi il linguaggio

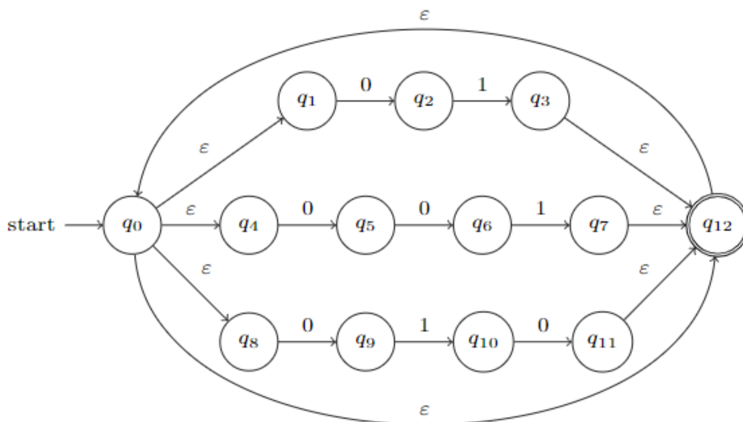
$$L_2 = \{w0^n \mid w \in \{0,1\}^* \text{ e } n = |w|\}.$$

Grammatica:
0^*0^n oppure 0^*1^n (lo rappresenterei come unione delle due)
$S_1 \rightarrow 0S_10 \mid \varepsilon$
$S_2 \rightarrow 0S_21 \mid \varepsilon$

<u>*Quarto appello 2017/2018*</u>

1. (a) Convertire l'espressione regolare $(\mathbf{01} + \mathbf{001} + \mathbf{010})^*$ in un ε-NFA (si può usare l'algoritmo visto a lezione oppure creare direttamente l'automa). **Importante:** l'automa deve essere nondeterministico e deve sfruttare le ε-transizioni per riconoscere il linguaggio.

Basta sapere le solite regole e la soluzione (tra le possibili) è:

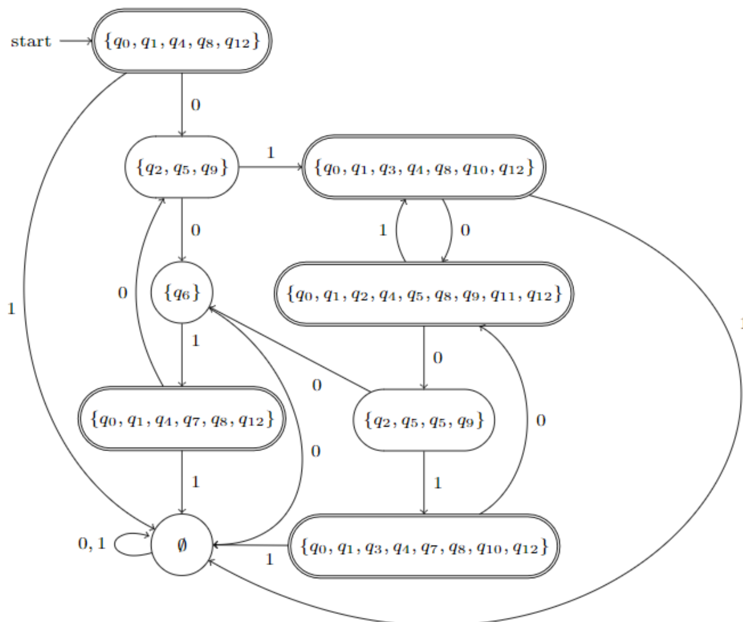


(b) Trasformare l'ε-NFA ottenuto al punto precedente in un DFA.

Considerando la ε-chiusura come stato iniziale: q0, q1, q4, q8, q12

q_i	0	1
→*{q0,q1,q4,q8,q12}	{q2,q5,q9}	∅
{q2,q5,q9}	{q6}	{q0,q1,q3,q4,q8,q10,q12}
{q6}	∅	{q0,q1,q4,q7,q10,q12}
*{q0,q1,q3,q4,q8,q10,q12}	{q0,q1,q2,q4,q5,q8,q9,q11,q12}	∅

*{q0,q1,q4,q7,q8,q12}	{q2,q5,q9}	∅
*{q0,q1,q2,q4,q5,q8,q9,q11,q12}	{q2,q5,q6,q9}	{q6}
{q2,q5,q6,q9}	{q0,q1,q3,q4,q7,q8,q10,q12}	{q6}
*{q0,q1,q3,q4,q7,q8,q10,q12}	{q0,q1,q2,q4,q5,q8,q9,q11,q12}	∅



Come sempre si ha l'inclusione della closure nel caso degli stati q3,q7,q11 e con pazienza si ricava tutto.

(a) Se i sta per la keyword if ed e sta per la keyword else, allora si chiede di definire una CFG capace di generare tutte le stringhe che rappresentano comandi if e if-else annidati e concatenati, come per esempio iii, iiee, iie, ieieiie e anche iieeiiee.

Un esempio di grammatica accettante è:
S → iS| iSeS | e

(si salta il secondo punto perché è banale, è più una risposta logica)

(c) La vostra grammatica del punto (a) è ambigua o no? Argomentate la risposta. In caso pensiate che sia ambigua, è possibile trovare un'altra CFG che generi lo stesso linguaggio e che non sia ambigua? Argomentate la risposta e se pensate che sia possibile, allora date una tale CFG non ambigua

La grammatica è sicuramente ambigua in quanto sono possibili più derivazioni, ad esempio:

19) S → iS → iiS → iiS → iiSeS
    (per ogni due i basta mettere "e" e vengono chiusi, due if corrispondono a due else)

oppure

20) S → iSeS → iiSeS
    (e anche qui si giunge alla stessa chiusura degli if)

Un esempio di grammatica non ambigua è:
S → iS | iS'eS   S' → e | iS'eS'

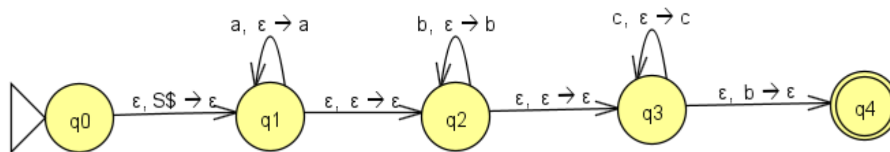
Considerate il linguaggio $L = \{www \mid w \in \{a,b\}^*\}$. Questo linguaggio è regolare? Dimostrare formalmente la risposta.

Il linguaggio non è regolare. Supponiamo per assurdo che lo sia:

- *sia h la lunghezza data dal Pumping Lemma;*
- *consideriamo la parola $v = a^h b a^h b a^h b$, che appartiene ad L ed è di lunghezza maggiore di h;*
- *sia $v = xyz$ una suddivisione di v tale che $y \neq \varepsilon$ e $|xy| \leq h$;*
- *poiché $|xy| \leq h$, allora xy è completamente contenuta nel prefisso a^h di v, e quindi sia x che y sono composte solo da a. Inoltre, siccome $y \neq \varepsilon$, possiamo dire che $y = a^p$ per qualche valore $p > 0$. Allora la parola xy^2z è nella forma $a^{2h+p} b a^h b a^h b$, e quindi non appartiene al linguaggio perché non è possibile suddividerla in tre sottostringhe uguali tra di loro.*

Abbiamo trovato un assurdo quindi L non può essere regolare.

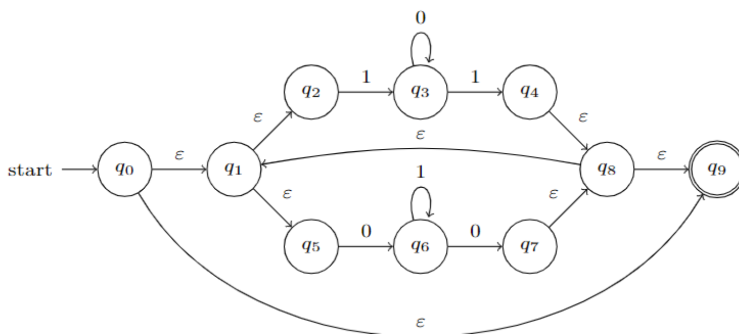
Descrivere un PDA che accetta per pila vuota ed è capace di riconoscere il linguaggio $L = \{(ab)^n (ca)^n \mid n \geq 1\}$. Il vostro è un automa deterministico o nondeterministico? Spiegare la risposta.



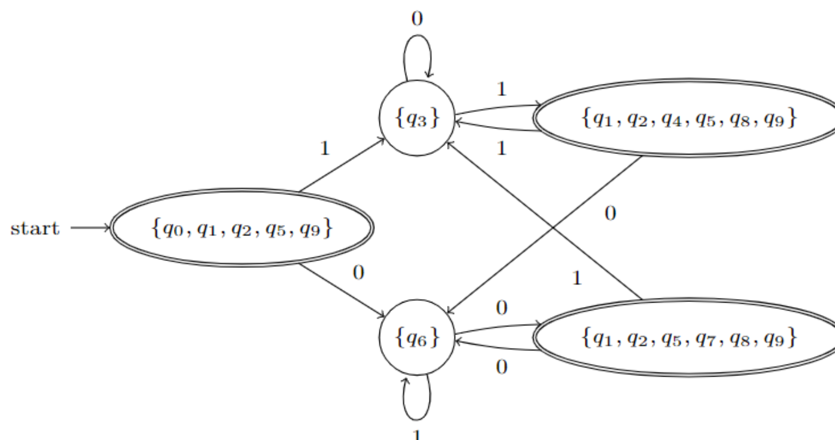
Si vede anche dal disegno; l'automa è a tutti gli effetti deterministico.

Terzo appello 2018

(a) Convertire l'espressione regolare $(\mathbf{10^*1} + \mathbf{01^*0})^*$ in un ε-NFA (si può usare l'algoritmo visto a lezione oppure creare direttamente l'automa). **Importante:** l'automa deve essere nondeterministico e deve sfruttare le ε-transizioni per riconoscere il linguaggio.



q_i	0	1
→*{q0,q1,q2,q5,q9}	q6	q3
q3	q3	{q1,q2,q4,q5,q8,q9}
q6	{q1,q2,q5,q7,q8,q9}	q6
		q3
		q3



Stabilire se il seguente linguaggio $L = \{a^n b^m c^k \mid n = k, n \geq 0, m \geq 0, k \geq 0\}$ è CF oppure no. Se pensate che sia CF, esibite una CFG che lo generi oppure un PDA che lo riconosca (spiegando perché questo è vero). Se pensate che non sia CF, date un argomento che dimostri che effettivamente non lo sia.

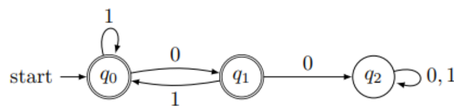
La grammatica precedente può essere descritta da:
S → aSc | B
B → bB | ε
Come si vede, risulta essere regolarmente context-free.

Considerate il linguaggio $L = \{0^{2n}1^m0^n : n, m \geq 0\}$. Questo linguaggio è regolare? Dimostrare formalmente la risposta.

Supponiamo per assurdo che L sia regolare, pertanto avremo la solita *k* come pumping length e consideriamo come parola w=$0^{2n}1^m0$.
Scegliendo poi come al solito y≠ε , |xy| <= k e una suddivisione w=xyz.
Come si vede anche dalla stringa stessa, il numero di 0 di z è determinato da k – 2n – (n) se (n) > 0
|xy| si può intendere tutto contenuto dentro 0^{3n} e quindi basta un pumping del tipo xy^3z per avere, partendo da un generico p > 0, $0^{3n+p}1^m$ dove già si può notare lo sbilanciamento degli 0 rispetto al numero di 1. Pertanto il linguaggio non può essere regolare.

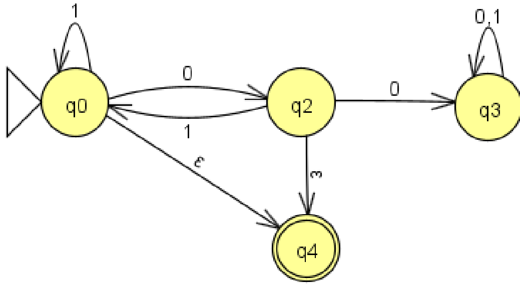
Prima parte luglio 2019

Considerate il DFA che riconosce il linguaggio di tutte le stringhe sull'alfabeto $\{0, 1\}$ che *non contengono* la sottostringa 00:

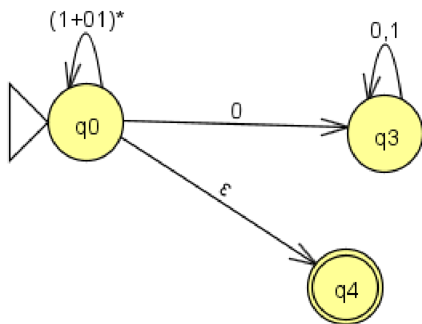


Trasformate il DFA in una Espressione Regolare usando l'algoritmo di eliminazione degli stati.

Siccome si può notare che lo stato iniziale ha più transizioni entranti e ci sono anche due stati finali, si seguono le solite regole, nello specifico: stati finali diventano stati non finali con ε-transizione verso il nuovo stato finale.



A questo punto cominciamo ad esempio eliminando q2.
Pre: q0 Succ: q3 (1+01)*0
Pre: q0 Succ: q4 (1+01)*+ ε



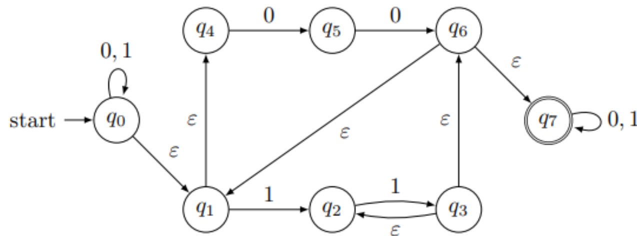
Come poi si vede eliminando q3, stato non finale:
Pre: q0 (1+01)*0

Per unione delle due combinazioni precedenti otteniamo come ER finale: (1 + 01)* + (1 + 01)* 0

Usate la soluzione dell'esercizio precedente per creare una Espressione Regolare che definisca il linguaggio di tutte le stringhe sull'alfabeto $\{0, 1\}$ tali che tutte le occorrenze di 11 appaiono prima di tutte le occorrenze di 00.

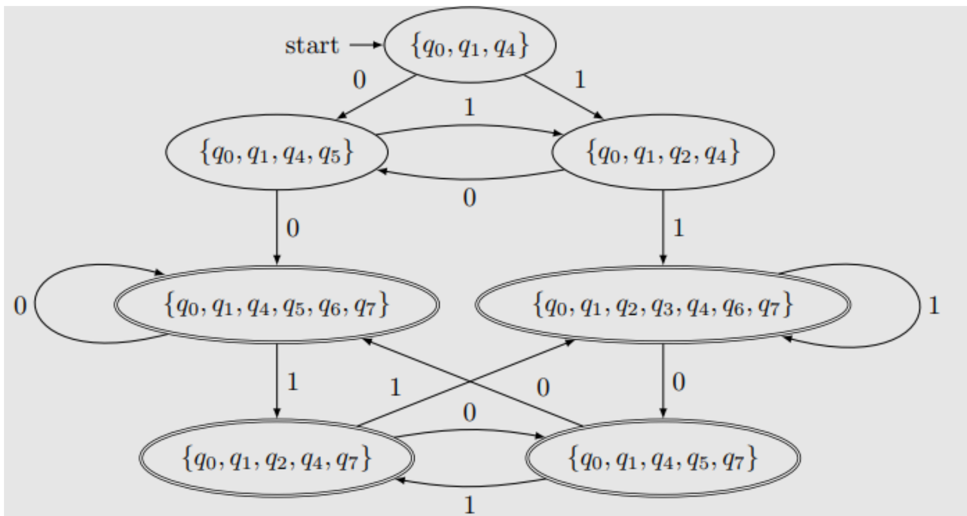
ER: (1+01)*(0+ε)(0+10)*(1+ε)
In pratica ragiono che rispetto prima l'inserimento di 11, poi stringa generica in mezzo, 00 e stringa generica in mezzo. Compattamente può essere scritta come: (1+01)*(0+10)*(1+ε)

Trasformate il seguente ε-NFA in DFA:



q_i	0	1
→{q0,q1,q4}	{q0,q1,q4,q5}	{q0,q1,q2,q4}
{q0,q1,q2,q4}	{q0,q1,q4,q5}	{q0,q1,q2,q3,q4,q6,q7}
{q0,q1,q4,q5}	{q0,q1,q4,q5,q6,q7}	{q0,q1,q2,q4}
*{q0,q1,q2,q3,q4,q6,q7}	{q0,q1,q4,q5,q7}	{q0,q1,q2,q3,q4,q6,q7}

*{q0,q1,q4,q5,q6,q7}	{q0,q1,q4,q5,q6,q7}	{q0,q1,q2,q4,q7}
*{q0,q1,q4,q5,q7}	{q0,q1,q4,q5,q6,q7}	{q0,q1,q2,q4,q7}
*{q0,q1,q2,q4,q7}	{q0,q1,q4,q5,q7}	{q0,q1,q2,q3,q4,q6,q7}



4. Sia $\Sigma = \{0, 1\}$ e considerate il linguaggio

$$LMN = \{0^\ell 1^m 0^n \mid \ell < n\}$$

(a) Completate il seguente schema di partita per il Gioco del Pumping Lemma in modo da far vincere il Giocatore 2:

Giocatore 1: sceglie il valore di h=7
Giocatore 2: sceglie la parola w = 000011111000000
Giocatore 1: suddivide la parola in x=000 y=011 z=111000000 quindi con |xy| <= z
Giocatore 2: esegue il pumping i=3 con w=000011011011111000000 che non appartiene al linguaggio originale. Dunque il linguaggio non è regolare.

(b) Dimostrate che LMN non è un linguaggio regolare usando il Pumping Lemma.
Considerata "h" la solita pumping length, consideriamo come parola $w=0^k1^p0^{k+1}$.
Operiamo la solita suddivisione della stringa w=xyz con:
$x=\varepsilon$ $y=0^k1^p$ $z=0^{k+1}$
e avremo successivamente:
xy^2z con $w=0^{2k}1^{2p}0^{k+1}$
avendo il resto della parola composto da 0 e per "p" parti della parola dagli 1.
Si nota però che con un pumping di questo tipo, il numero degli 0 della prima parte sbilancia quelli della seconda parte ed il linguaggio non è regolare.

Si consideri la seguente grammatica CF, G: $S \to aBb$, $B \to aBb \mid BB \mid \varepsilon$

(a) Descrivere il linguaggio $L(G)$ in termini di un linguaggio L composto da "stringhe w con certe proprietà".

a) La soluzione cita il linguaggio delle parentesi bilanciate, ma oltre a questo, per quello che ne sappiamo noi, possiamo definire il linguaggio come quello delle a/b alternate, più formalmente definibile come a(w')b = a(w'')b notando quindi come la stringa a sia sempre prefisso, possibilmente anche definita come prefisso "ab".

(b) Dimostrare per induzione che tutte le stringhe in $L(G)$ sono in L, cioè che $L(G) \subseteq L$.

b) Induttivamente, seguiamo sempre la strada delle ab alternate/prefisso, tale che se le stringhe sono vuote chiaramente il linguaggio è L. Al passo induttivo, quindi ad i+1, avremo che il linguaggio ha prefisso "a" e successivamente il linguaggio può essere composto da derivazioni del tipo "ab" oppure "bb" tali che avremo ben definita la grammatica che stiamo derivando, quindi w'. w'' similmente è ben definita sulle stringhe alternate, in quanto composta ricorsivamente da un'alternanza stessa della precedente ed entrambe le stringhe, per successiva proprietà di concatenazione definita chiusa, è induttivamente corretta e quindi sono nel linguaggio.

Prima parte 19 Settembre 2019

Costruire una CFG G che genera il linguaggio $L = \{a^n b^m c^k \mid \ con\ n = m\ o\ m = k\ e\ n, m\ e\ k \geq 0\}$.
Dimostrare che per la grammatica G che proponete, vale $L(G) \subseteq L$.

Una grammatica per questo linguaggio può essere:
S → AB | BC
A → aAb | ε
B → aB | bCb | ε
C → cC | ε

Per la grammatica ottenuta si ha che il linguaggio è valido e regolare. Induttivamente possiamo considerare che valgono due casi, nello specifico $a^n b^n$ oppure $b^n c^n$, mettendo per comodità un unico esponente per meglio rappresentare le condizioni date dall'esercizio. In tali condizioni, il caso base n=0 porta ad una chiara regolarità (ab oppure bc), mentre per il caso induttivo considera che abbiano entrambi lo stesso numero di lettere. Per concatenazione quindi $a^n b^m c^k$ risulta bilanciato sulla base degli b, avendo quindi il numero di a pari al numero di b e similmente il numero di c pari al numero di b. Il linguaggio permane regolare ed è anche CF quindi date le citate considerazioni.

Sia $\Sigma = \{0, 1\}$ e considerate i due seguenti linguaggi:

$$L_1 = \{(01)^n 0 (10)^n \mid n \geq 0\}$$
$$L_2 = \{1^n 0 1^n \mid n \geq 0\}$$

Uno dei due linguaggi è regolare, l'altro linguaggio non è regolare.

(a) Dire quale dei due linguaggi è regolare e quale non è regolare.
(b) Per il linguaggio regolare, dare un automa a stati finiti o un'espressione regolare che lo rappresenta.
(c) Per il linguaggio non regolare, dimostrare la sua non regolarità usando il Pumping Lemma.

a) Si nota che il primo linguaggio è composta dalla possibile ripetizione della sottostringa 01, come tale può essere ad occhio riconoscibile come ER/automa (abbiamo visto tanti casi simili, non saprei come altro spiegarlo, si vede). Nel caso L_2 invece si nota sempre ad occhio che il numero degli 1 è sempre maggiore rispetto al numero degli 0 e quindi non è regolare.
b) Il primo linguaggio, banalmente, è rappresentabile come: (01)*0
c) Dando le solite condizioni del PL, y ≠ 0 , xy <= p e $xy^iz \in L$
Prendendo come parola $1^p 0 1^n$
Avremo sicuramente un esponente "p" > 0 | saremo nelle condizioni $w=1^{n+p}01^p$
notando quindi lo sbilanciamento degli 1 rispetto agli zeri nella sottostringa xy.
Il linguaggio quindi non è regolare.

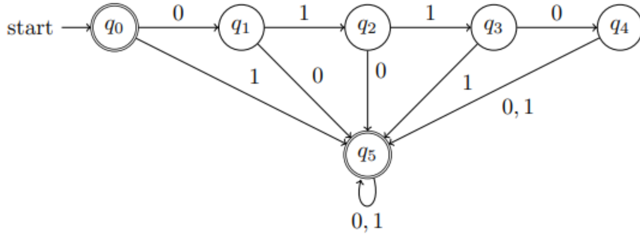
Scrivere un automa a stati finiti che riconosca il linguaggio

$$L = \{w \in \{0, 1\}^* \mid w \neq 0110\}$$

L'osservazione da fare è, avendo il caso dell'automa che accetta tutto *tranne* 0110, di crearsi l'automa che accetta solo la stringa 0110 e farne il complementare.
Per non appesantire la rappresentazione sintetizzo facilmente i passaggi:

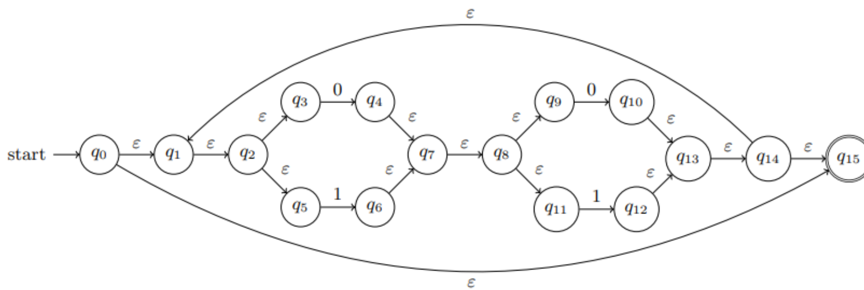
21) si prende l'automa e si costruisce quello che accetta solo 0110
22) si complementa l'automa (passando a 4 stati finali e quello non finale)
23) si costruisce l'automa finale, con 5 stati non finali ed un unico stato finale in cui, da ogni stato, si avanza verso lo stato finale con lo stato che non sta usando in quel momento (avanzo con 1, necessariamente andrò a 0 verso lo stato finale e viceversa).

Lo spiego a parole per completezza, se non è chiaro basta farselo e convincersi.
Risultato:



Trasformare l'espressione regolare $((0+1)(0+1))^*$ in un automa *usando l'algoritmo visto a lezione.*

Quindi:

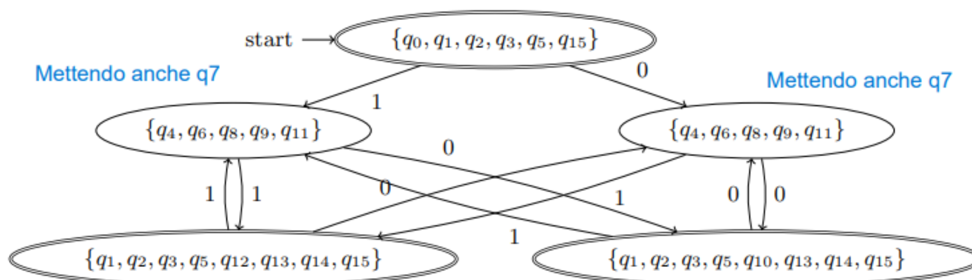


Trasformare l'ε-NFA ottenuto nell'esercizio 2 in DFA

Nota: nella soluzione del prof non considera q7 nelle transizioni, ma direi che non è corretto visto che le ε-transizioni ragionando induttivamente e bisogna mettere tutti gli stati raggiunti.
Tipo per lo stato iniziale mette q8,q9,q11, per lo stesso motivo dovrò considerare q4,q6,q7 per chiusura, visto che il simbolo 0 l'ho già consumato prima.

q_i	0	1
→*{q0,q1,q2,q3,q5,q15}	{q4,q6,q7,q8,q9,q11}	{q4,q6,q7,q8,q9,q11}
{q4,q6,q7,q8,q9,q11}	{q1,q2,q3,q5,q10,q13,q14,q15}	{q1,q2,q3,q5,q12,q13,q14,q15}
*{q1,q2,q3,q5,q10,q13,q14,q15}	{q4,q6,q7,q8,q9,q11}	{q4,q6,q7,q8,q9,q11}
*{q1,q2,q3,q5,q12,q13,q14,q15}	{q4,q6,q7,q8,q9,q11}	{q4,q6,q7,q8,q9,q11}

Siccome è un DFA, va rappresentato con una sola transizione che va da ogni stato ad un altro, ragion per cui uno di questi si ripete.

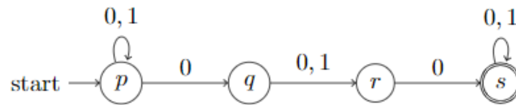


Primo compitino 2017/2018

Considerare il linguaggio $L = \{$stringhe di a e b che iniziano con a e finiscono con $a\}$

(a) Dare un automa a stati finiti *deterministico* che accetti il linguaggio L.
(b) Dare un'espressione regolare che rappresenti il linguaggio L.

Dato il seguente NFA



costruire un DFA equivalente

Il linguaggio

$$L = \{a^n b^m c^{n-m} : n > m > 0\}$$

è regolare? Motivare in modo formale la risposta.

Sia L un linguaggio regolare su un alfabeto Σ. Dimostrare che anche il seguente linguaggio è regolare:

$$init(L) = \{w \in \Sigma^* : \text{ esiste } x \in \Sigma^* \text{ tale che } wx \in L\}$$

Prima prova intermedia worskhop 2020-2021

Considera la funzione ricorsiva

$$faro(x, z) = \begin{cases} z & \text{se } x = \varepsilon \\ a.faro(z, y) & \text{se } x = ay \end{cases}$$

Per esempio, $faro(00110, 0101) = 000110110$. Dimostra che se L e M sono linguaggi regolari definiti sullo stesso alfabeto Σ, allora anche il linguaggio $faro(L, M) = \{faro(x, z) \mid x \in L \text{ e } z \in M\}$ è regolare.

Abbiamo che il linguaggio ricorsivamente dice:

1) se x è vuota, avrò "z" come stringa
2) se x è uguale a due pezzi della parola, allora ci sarà una concatenazione tra a e la funzione stessa richiamata ricorsivamente su "z"
3) A quel punto possiamo avere x=ε come caso 1
4) altrimenti x=ay e richiamo ricorsivo.

A queste condizioni sappiamo che se "x" è più corta di "z", "x" alterna simboli con "z" finché possibile e poi continua con la parte rimanente di "z", altrimenti avremo l'alternanza degli stessi simboli ma al contrario.
L'automa quindi pone delle triple di transizione in cui avremo come stati, oltre ad L linguaggio stesso, almeno due stati (fasi ricorsive appena vista) tali da rappresentare come triple gli stati dell'automa.
Quindi lo stato A può essere definito come (r_L, r_M, L) e le transizioni sono definite in questo modo (ci sono due stati potenzialmente, come visto prima dalla descrizione e quindi):

1) $(r_L, r_M, L) \rightarrow (s_L, r_M, M)$ se $\delta_L(r_L, a)=s_L$, dunque si usa la prima delle transizioni, cambiando il turno per la successiva
2) $(r_L, r_M, M) \rightarrow (r_L, s_M, L)$ se $\delta_L(r_M, a)=s_M$, dunque si usa la seconda delle transizioni, invertendo le carte in tavola

A questo punto occorre stabilire chi sono gli stati finali; dunque, possiamo avere dei casi in cui abbiamo l'alternanza di L, M e la possibilità che entrambi siano non finali nella computazione.
Dunque, avremo due situazioni similari a prima in cui si alternano L ed M e altri due risultati che rappresentano l'altra possibile computazione sulla parola in L o in M nel caso terminasse prima.

Scritto da Gabriel

- $(r_L, r_M, L) \xrightarrow{a} (s_L, r_M, L_{suff})$ se $\delta_L(r_L, a) = s_L$ e r_M è uno stato finale di A_M;
- $(r_L, r_M, M) \xrightarrow{a} (r_L, s_M, M_{suff})$ se $\delta_M(r_M, a) = s_M$ e r_L è uno stato finale di A_L;
- $(r_L, r_M, L_{suff}) \xrightarrow{a} (s_L, r_M, L_{suff})$ se $\delta_L(r_L, a) = s_L$;
- $(r_L, r_M, M_{suff}) \xrightarrow{a} (r_L, s_M, M_{suff})$ se $\delta_M(r_M, a) = s_M$.

A questo punto, si può descrivere come possibile stato iniziale la tripla (q_L^0, q_M^0, L) e abbiamo che il linguaggio è effettivamente regolare.

Considera il linguaggio

$$L_2 = \{w \in \{1, \#\}^* \mid w = x_1 \# x_2 \# \dots \# x_k \text{ con } k \geq 0, \text{ ciascun } x_i \in 1^* \text{ e } x_i \neq x_j \text{ per ogni } i \neq j\}.$$

Dimostra che L_2 non è regolare.

Come sempre, supponiamo per assurdo il linguaggio sia regolare e avremo un generico "k" come pumping length. Stanti queste condizioni (il cancelletto # è un simbolo alternato tra gli 1 presenti, ciascuno con un indice diverso come si può leggere dalla richiesta del linguaggio) definiamo come parola:
$w=1^p\#1^{p+1}\#1^{p+2}$.... $\#1^{p+k}$
Sapendo che le condizioni sono:

3) $y \neq \varepsilon$
4) $|xy| <= k$

ad esempio, con un pumping p=0, che nella parte cosiddetta finale (z) della parola avremo un numero di maggiore, in quanto presente almeno una serie di 1 del tipo 1^{p+k-q}, intendendo per q la potenza usata per avere un pumping nella parte iniziale. La parola xy^0z non apparterrà al linguaggio e dunque questi non può dirsi regolare.

Considera una generalizzazione delle grammatiche context-free che consente di avere espressioni regolari sul lato destro delle regole di produzione. Senza perdita di generalità, puoi assumere che per ogni variabile $A \in V$, la grammatica generalizzata contenga un'unica espressione regolare $R(A)$ su $V \cup \Sigma$. Per applicare una regola di produzione, scegliamo una variable A e la sostituiamo con una parola del linguaggio descritto da $R(A)$. Come al solito, il linguaggio della grammatica generalizzata è l'insieme di tutte le stringhe che possono essere derivate dalla variabile iniziale.

Per esempio, la seguente grammatica generalizzata descrive il linguaggio di tutte le espressioni regolari sull'alfabeto $\{0, 1\}$. I simboli in rosso sono terminali, i simboli in nero sono variabili oppure operatori.

$S \to (T+)^*T + \emptyset$	(Espressioni regolari)
$T \to \varepsilon + F^*F$	(Termini = espressioni che si possono sommare)
$F \to (0 + 1 + (S))(* + \varepsilon)$	(Fattori = espressioni che si possono concatenare)

Dimostra che ogni grammatica context-free generalizzata descrive un linguaggio context-free. In altre parole, dimostra che consentire espressioni regolari nelle regole di produzione non aumenta il potere espressivo delle grammatiche context-free.

Per dire senza perdita di generalità che una grammatica context-free generalizzata sia effettivamente linguaggio context-free, stiamo fondamentalmente dicendo che le espressioni regolari presenti nelle regole e poi ottenute per derivazione siano date dall'unione delle precedenti.
Questo descrive una grammatica generalizzata, la quale deve essere context-free.
Per poterla considerare context-free, deve innanzitutto essere sempre verificata che esiste una sola regola terminale, tutte le altre sono del tipo A → u, intendo per "u" un generica stringa espressiva dove finisce il linguaggio. Ad esempio qui:

$S \to (T+)^*T + \emptyset$	(Espressioni regolari)
$T \to \varepsilon + F^*F$	(Termini = espressioni che si possono sommare)
$F \to (0 + 1 + (S))(* + \varepsilon)$	(Fattori = espressioni che si possono concatenare)

si vede che dovremo rimpiazzare

1) l'operazione di unione (+) del tipo A → B + C con A → B ed A → C (in questo caso quindi ne avremmo diverse, ad esempio con S → T, S → T*, T →F*F, ecc.)
2) l'insieme vuoto, che significa non stiamo facendo nessun avanzamento, quindi sarà un semplice S → S
3) le ε-regole, le quali saranno sostituite dalle corrispettive espressioni (tipo T → ε, quindi rimpiazzeremo nelle altre F*F)
4) la * di Kleene, formato da tutte le possibili espressioni della stringa, precisamente A → ε, A_S → S, A→A_SA
5) la concatenazione, quindi come prima per l'unione partiamo da A → R.S che dunque è formata da A→$A_R A_S$, splittando poi in A→R ed A→S

Facendo tutte queste operazioni sapremo che il linguaggio sia ricostruibile in maniera standard (usando quindi regole di Chomsky e dei CF in sè per sè).

Terzo appello 2021

(8 punti) Considera il linguaggio

$$L = \{0^m 1^n \mid m/n \text{ è un numero intero}\}.$$

Dimostra che L non è regolare.

Supponiamo che per assurdo L sia regolare e, considerando "k" come Pumping Length, considerando la parola $w=0^{k+1}1^{k+1}$ tale che il rapporto dia come minimo 1.
Scegliendo poi una suddivisione w = xyz, tale che y≠ε, w=xyz e |xy| <= k, avremo che per p, q > 0, avremo con pumping del tipo xy^0z delle stringhe del tipo $w=0^q0^{k+1-q-p}0^{k+1}$ tale che il numero di 0 sia sbilanciato, quindi maggiore rispetto al numero possibile di 1. Quindi il linguaggio non è regolare.

2. **(8 punti)** Per ogni linguaggio L, sia prefix$(L) = \{u \mid uv \in L$ per qualche stringa $v\}$. Dimostra che se L è un linguaggio context-free, allora anche prefix(L) è un linguaggio context-free.

Se L è un linguaggio CF, esiste una grammatica in forma normale di Chomsky che la genera.
Siccome dobbiamo generare la grammatica dei prefissi, necessariamente ad una variabile V corrisponderà una successiva variabile V' dove V rappresenta il prefisso di tale parola del linguaggio.
Come tale la forma normale di Chomsky deve avere regole del tipo:
A→BC
A→a
ove esisteranno delle regole necessariamente strutturabili come:
S' → S
S → ASA
A→ASB
B → b
dove si può notare una strutturazione tale da garantire l'alternanza di almeno due simboli, ricorsivamente, fino ad un terminale.
Per ogni singola regola, ognuna di queste appartiene anche alla grammatica generata G' e similmente anche le variabili S' → S ed S' → ε.
Ad esso apparterranno anche le regole unitarie, quindi regole del tipo V' → AB' con V → A' e anche la stessa variabile iniziale S che sarà S' per G da G'.
Date tutte queste caratteristiche (l'idea di derivazione di grammatica che ho suggerito è indicativa), il linguaggio risulta CF.

Terzo appello 2019 – 5 Settembre 2019

Descrivere, anche tramite un disegno, la relazione che esiste tra i linguaggi regolari, i linguaggi CF, i linguaggi CF non inerentemente ambigui, i linguaggi riconosciuti dai PDA che accettano per stato finale, quelli riconosciuti dai PDA che riconoscono per pila vuota e, per finire, quelli riconosciuti dai DPDA che accettano per stato finale e quelli riconosciuti dai DPDA che accettano per pila vuota. La risposta va motivata in modo conciso.

Tutte queste caratterizzazioni sono composte da linguaggi che accettano un insieme comune di operazioni chiuse, descrivendo principalmente unione, concatenazione, chiusura di Kleene, ecc.
Le varie operazioni sono definite induttivamente, tali che:

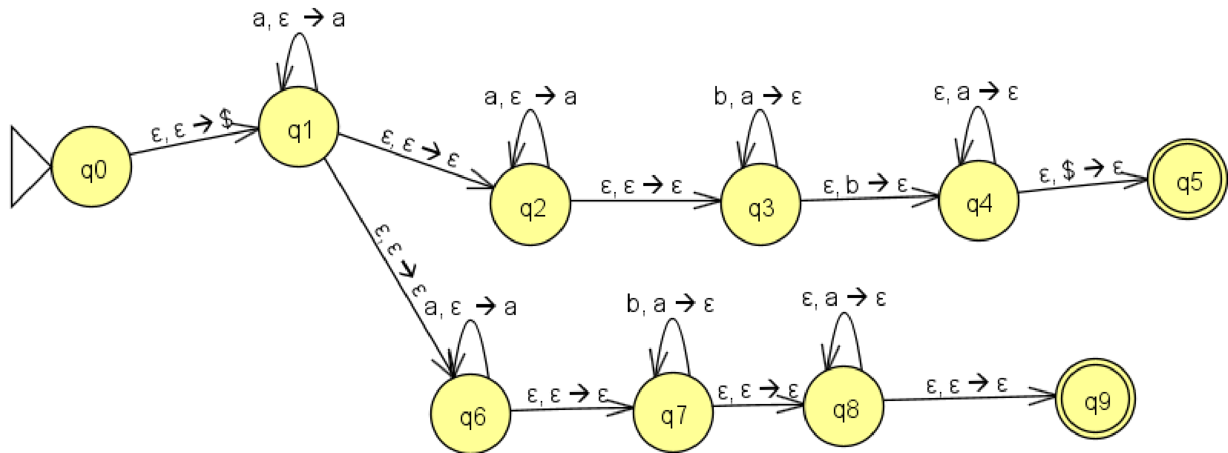
1) nei linguaggi regolari danno luogo ad un automa, tale che esista almeno un ciclo (PL)
2) nei linguaggi CF, analogamente, esiste una sottoderivazione ricorsiva (PL per i CF)
3) nei linguaggi per PDA, essi avranno necessariamente uno stato iniziale, svuotato alla fine della pila ed un insieme di stati finali, tali che le transizioni regolari siano composte da push/pop degli stati tolti alla fine (pila vuota).

Tutti quindi presentano stati ripetuti, sceglibili deterministicamente/indeterministicamente, tali da poter avere ripetizione di stringhe e scelta possibile a seconda delle proprietà del linguaggio.

2. Descrivere un PDA, che accetta per pila vuota, che riconosca il seguente linguaggio $L = \{a^n b^m \mid 0 \leq n \leq m \leq 2n\}$. E' possibile costruire il PDA passando prima per una CFG che genera L. In questo caso è richiesta una dimostrazione o almeno una spiegazione convincente del fatto che la CFG generi veramente L.

Il linguaggio è costituito da un numero diverso di "a" e "b", avendo in particolare un numero doppio come massimale di "a" rispetto a quello delle "b".
L'ordine di queste stringhe deve essere mantenuto, avendo idealmente un numero di "a" uguale a quelle delle "b" facendo pop di un singolo per volta, oppure eseguendo il pop di due "b" e ottenendo m=2n oppure alternando le cose, avendo quindi un numero doppio di "a" rispetto alle "b" nelle normali condizioni.
Partiamo dalla descrizione di una grammatica che descrive questo linguaggio:
S → aA | ε
A → aAbB | ε
B → aB | ε
In questo modo posso avere già almeno un numero doppio di "a" rispetto alle "b" e la CFG genera effettivamente questo linguaggio, avendo ad esempio come derivazione:
S → aA → aAbB → aAbaB → aAbBaB → aab

Segue il possibile PDA di questa grammatica:
(seguendo l'idea numero di "a"=numero di "b" oppure per l'idea di prima numero doppio di "a" rispetto al numero di "b")



Si chiede di descrivere la Forma Normale di Chomsky (FNC) e di descrivere l'enunciato del pumping Lemma, spiegando, infine, come il fatto che ogni CFG possa venire messa in FNC sia fondamentale per dimostrare il pumping Lemma.

La forma normale di Chomsky si pone l'idea di avere regole formate da derivazioni con al massimo 2 transizioni, di cui ve ne è almeno una terminale. Essendo che ogni linguaggio context-free è dimostrabile essere generato da una formale normale di Chomsky, ci si pone l'obiettivo di trasformarlo:

4) aggiungendo una nuova variabile iniziale
5) eliminazione delle ε-regole
6) eliminazione regole unitarie A → B
7) trasformazione delle regole rimaste per avere solo 2 transizioni finali

Similmente, il pumping lemma si propone, nel caso dei linguaggi CF, di avere una parola di lunghezza |w| >= k con una suddivisione w=uvxyz tali che possiamo pompare uv^ixy^iz per un i >=0
Si nota quindi che il PL sia dimostrato a causa del fatto che, essendo una grammatica ricorsiva, stanti le condizioni di prima abbiamo bisogno di almeno 2 simboli ripetuti ed 1 terminale, avremo almeno una singola ripetizione delle parole (seguendo l'idea del $B=2^V$ +1 del pumping lemma, nel caso in cui almeno uno dei due simboli sia terminale, pur avendo regole vuote, da cui il +1)
Ecco quindi la stretta correlazione tra i due.

<u>05/07/2021: Prima parte 2 Appello 2021</u>

1. Considera la seguente funzione da $\{0,1\}^*$ a $\{0,1\}^*$:

$$\text{stutter}(w) = \begin{cases} \varepsilon & \text{se } w = \varepsilon \\ aa.\text{stutter}(x) & \text{se } w = ax \text{ per qualche simbolo } a \text{ e parola } x \end{cases}$$

Dimostra che se L è un linguaggio regolare sull'alfabeto $\{0,1\}$, allora anche il seguente linguaggio è regolare:

$$\text{stutter}(L) = \{\text{stutter}(w) \mid w \in L\}.$$

2. Considera il linguaggio

$$L_2 = \{w0^n \mid w \in \{0,1\}^* \text{ e } n = |w|\}.$$

Dimostra che L_2 non è regolare.

3. Per ogni linguaggio L, sia $\text{suffix}(L) = \{v \mid uv \in L \text{ per qualche stringa } u\}$. Dimostra che se L è un linguaggio context-free, allora anche $\text{suffix}(L)$ è un linguaggio context-free.

Se L è un linguaggio regolare esiste un DFA che riconosca L.
Avremo quindi una grammatica G (Q, ∑, δ, q_0, S} per cui dall'automa sarà riconosciuta similmente G {Q', ∑, δ,' q_0, S'}
Se applicassimo stutter su (0110) avremmo la parola 00111100, letteralmente invertito
Si nota che il linguaggio può essere composto da:

1) ε nel caso in cui w = ε
2) ricorsivamente aa.stutter, nel caso in cui w = ax nell'alfabeto {0,1}*

L'automa quindi procede ripetendosi solamente se lo stato attuale è uguale a quello precedente, in particolare concatenando la parte ricorsiva (aa), la parola (x) e w (stringa attuale).
Ragioniamo quindi creando degli stati A formati da triple (aa, r_X, r_w).
Avremmo quindi due possibili casi:

1) (aa, r_X, r_w) → (aa, r_x) nel caso in cui w= ε (rimane inalterato)
2) (aa, r_X, r_w) → (aa, s_X, s_w) con s_w=ax nel caso in cui invece si abbia w = ax, in questo modo (x) potrà essere parola generica, ci sarà (aa) ripetuto ricorsivamente e l'altro simbolo rappresenta un simbolo nell'alfabeto {0,1}*.

Stabilite le possibili transizioni, definiamo gli stati finali dell'automa.
L'idea del linguaggio è avere alternanza di "a" ed "x" in maniera discontinua; quindi, gli stati finali saranno idealmente costituiti da alternanza di entrambi i simboli.
Al di là del caso di w=ε, avremmo una situazione in cui avremo:

1) F_A=aa.(s_w, s_x, w) in cui appunto si ha la ripetizione di aa concatenato all'alternanza di "w", di "x" e della stessa parola w scelta nell'alfabeto
2) F_A=aa.(s_x s_w, w) in cui si hanno gli stessi stati letteralmente alternati gli uni con gli altri nell'ordine inverso.

L'idea del linguaggio è letteralmente di duplicare stringhe, alternandole. Con questa dimostrazione si nota la regolarità del linguaggio.

2) Assumiamo per assurdo il linguaggio sia regolare.
Per completezza assumo entrambi i casi (a noi serve |w| = n)

1) se n ≠ w → 0^k0^p per k, p > 0.
   Assumiamo quindi w=xyz → 0^k0^p
   Con pumping xy^0z → x=0^p y=0^q z=0^{k-p-q}
   Quindi avremo uno sbilanciamento degli 0 nelle parti della parola, infatti:
   0^p0^{k-p-q} → 0^{k-q} che sarebbe assurdo

Dunque, il linguaggio non è regolare.

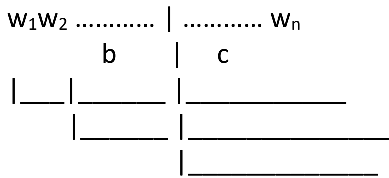
2) se n=w, 1^k0^p per k, p > 0.
   A questo punto assumiamo (date le solite condizioni y ≠ ε, |xy| <= k):
   w=xyz → 1^k0^p
   Con pumping xy^0z, il rimanente numero di 0 sarà compensato dagli 1 del resto della parola, quindi:
   x=1^p y=1^q z=$1^{k-p-}0^k$
   Quindi avremo più 1 che 0, infatti:
   $1^p1^{k-p-q}0^k$ → $1^{k-q}0^k$ che sarebbe assurdo

Dunque, il linguaggio non è regolare.

3) (fatto così dal prof)

Sia G la grammatica che genera L.
Assumiamo che G sia in forma normale di Chomsky.
Avremo quindi regole del tipo:
A → BC
D → d

Siccome c'è una concatenazione, un pezzo della parola sarà generato da "b" e un altro da "C"



A’ suffissi delle parole generate da A
A’ → A|B’C|C|C’|ε

Quindi quello che viene adesso è la variabile iniziale
D’ → d|ε
S’ è la variabile iniziale sse S è variabile iniziale di G

L’esercizio dei *prefix* inverte la stessa cosa.

<u>IV Appello 2021</u>

1. **(8 punti)** Considera il linguaggio
$$L = \{0^m 1^n \mid m > 3n\}.$$
Dimostra che L non è regolare.

2. **(8 punti)** Per ogni linguaggio L, sia substring$(L) = \{v \mid uvw \in L$ per qualche coppia di stringhe $u, w\}$. Dimostra che se L è un linguaggio context-free, allora anche substring(L) è un linguaggio context-free.

1) Assumiamo per assurdo il linguaggio sia regolare e immaginiamo “h” come possibile pumping length.
Immaginiamo, date le condizioni y≠ε, w=xyz, |xy| <= k,
una suddivisione $k=0^{3p+1}1^p$ con p > 0
Consideriamo quindi un pumping $w=xy^iz$ dove i=0
e la parola sarebbe $w=0^{3p+1}1^p1^{k-p}$ → $0^{3p+1}1^k$ che non appartiene al linguaggio e quindi il linguaggio non è regolare.

2) Se L è linguaggio CF, allora esiste una grammatica che lo genera. Possiamo assumere che G sia in forma normale di Chomsky e, come tale, ammetterà sempre regole del tipo A → BC con A,B,C simboli non terminali ed A → b simbolo non terminale.
Quindi nella normale grammatica fatta ad esempio:

A → BC
A → b

essendo la proprietà da mantenere la sottostringa (quindi “u” e “w” due stringhe prima e dopo di “v”):
A → BC
B → aBb | D
D → c
C → b

Infatti, un esempio di derivazione sarebbe (leftmost):
A → BC → aBbC → aaBbbC → aaDbbC → aacbbb

V Appello 2021

1. (8 punti) Considera il linguaggio

$$L = \{0^m 1^n \mid m = n^3\}.$$

Dimostra che L non è regolare.

2. (8 punti) Per ogni linguaggio L sull'alfabeto Σ, sia superstring$(L) = \{xyz \mid y \in L \text{ e } x, z \in \Sigma^*\}$. Dimostra che se L è un linguaggio context-free, allora anche superstring(L) è un linguaggio context-free.

1) Assumiamo per assurdo il linguaggio sia regolare e immaginiamo "h" come possibile pumping length.
Immaginiamo, date le condizioni y≠ε, w=xyz, |xy| <= k,
una suddivisione $k=0^{3p}1^p$ con p > 0
Consideriamo quindi un pumping $w=xy^iz$ fatto per una generica potenza k > 0, dove avremmo una stringa del tipo $s'=0^{3p+|y|}1^p$. In queste condizione, un qualsiasi pumping non sarebbe chiaramente nel linguaggio.
Anche nel caso di i=0, dove avremmo invece
$w=0^{3p}1^p1^{k-p}$ → $0^{3p}1^k$ sempre non appartenente ad L e quindi il linguaggio non è regolare.

2) Se L è linguaggio CF, allora esiste una grammatica che lo genera. Possiamo assumere che G sia in forma normale di Chomsky e, come tale, ammetterà sempre regole del tipo A → BC con A,B,C simboli non terminali ed A → b simbolo non terminale.
Quindi nella normale grammatica fatta ad esempio:

A → BC
A → b

essendo la proprietà da mantenere la superstringa (quindi avremo "u" simbolo ed "x", "y" due simboli qualsiasi facenti parte del linguaggio):
A → BC
B → aBc | D
C → cC | E
D → b
E → c

Infatti, un esempio di derivazione sarebbe (leftmost):
A → BC → aBcC → aDcC → abcC → abcE → abcc

Esercizi Macchine di Turing e Varianti di Macchine di Turing

Dal link di esercizi https://app.wooclap.com/MRRPGH/questionnaires/6268fb75de638d1394d4317b

Per le prime 5 risposte ci si riferisce a : https://turingmachine.io/?import-gist=4b36a1dc3fbff6e3a5509fe8867f8ca7
Cosa fa questa macchina di Turing con input 000111?
Risposta: Termina la computazione nello stato accept.

Cosa succede con input 000000?
Risposta: Va in loop

Qual è il linguaggio riconosciuto dalla Turing Machine?
Risposta: Tutte le parole che contengono almeno un 1

Questa Turing Machine è un decisore?
Risposta: No

Trasforma la Turing Machine in un decisore che riconosce lo stesso linguaggio.
Inserisci il codice della nuova macchina nella risposta.
Risposta:

```
input: '000111'
blank: ' '
start state: right
table:
  right:
    0: R
    1: {R: accept}
    ' ': {L: left}
  left:
    0: L
    1: {L: reject}
    ' ': {R: right}
  accept:
  reject:
```

Modifica la Turing Machine in modo che decida il linguaggio delle sequenze di 0 che sono potenze di 3: $\{0^{3^n} \mid n >= 0\}$
TM di riferimento: https://turingmachine.io/?import-gist=56dcb0347f17a48392f670c5c5009cd7
Risposta:

```
input: '000000'
blank: ' '
start state: start
table:
  start:
    0: (write: ' ', R: one}
    ' ': {R: reject}
  one:
    x: R
    0: {write: x, R: two}
    ' ': {R: accept}
  two:
```

```
    x: R
    0: (write: x, R: three}
    ‘ ‘: {R: reject}
  three:
    x: R
    0: {R: jump}
    ‘ ‘: {L: back}
  jump:
    x: R
    0: {write: x, R: two}
    ‘ ‘: {R: reject}
  back:
    [0, x]: L
    ‘ ‘: {R: one}
  accept:
  reject:
```

Modifica la Turing Machine in modo che decida il linguaggio delle coppie di parole dove la seconda stringa è il complemento a 1 della prima (0 e 1 scambiati), per esempio '1101#0010'
TM di riferimento: https://turingmachine.io/?import-gist=1c8d8998e5fe85b20f17fb4fb55b935e
Risposta:

```
input: '1101#0010'
blank: ‘ ‘
start state: start
table:
  start:
    [0,1]: R
    ‘#’: {L:  rollback}
    ‘ ‘: {R: reject}
  rollback:
    [0,1]: L
    ‘ ‘: {R: check}
  check:
    0: (write: 'x', R: zero}
    1: {write: ‘x', R: one}
    ‘#': {R: continue}
  zero:
    [0,1]: R
    '#': (R: is_one}
  one:
    [0,1]: R
    ‘#': (R: is_zero}
  is_zero:
    ‘x': R
    0: {write: 'x', L: back1}
  is_one:
    ‘x’: R
    1: {write: 'x', L: back1}
  back1:
    ‘x': L
    ‘#': {L: back2}
  back2:
    [0,1]: L
```

Scritto da Gabriel

'x': {R: check}
continue:
'x': R
' ': {R: accept}
[0,1]: {R: reject}
accept:
reject:

Modifica la Turing Machine in modo che decida il linguaggio delle parole con lo stesso numero di 0 e di 1.
Per esempio, accetta '001110' ma rifiuta '10010'.
TM di riferimento: https://turingmachine.io/?import-gist=1c8d8998e5fe85b20f17fb4fb55b935e
Risposta:

```
input: '001110'
blank: ' '
start state: start
table:
 start:
 #inizio con 0 oppure con 1
  0: {write: x, R: caso0}
  1: {write: y, R: caso1}
  #caso base = stringa vuota
  '': {R: accept}
  x: R
  y: R
 caso0:
  0: R
  1: {write: y, L: match}
  [y, ' ']: {R: reject}
 caso1:
  0: {write: x, L: match}
  1: R
  [x, ' ']: {R: reject}
 match:
  0: {write: x, L: start}
  1: {write: y, L: start}
  ' ': {R: reject}
 accept:

 reject:
```

A3.10 Say that a ***write-once Turing machine*** is a single-tape TM that can alter each tape square at most once (including the input portion of the tape). Show that this variant Turing machine model is equivalent to the ordinary Turing machine model. (Hint: As a first step, consider the case whereby the Turing machine may alter each tape square at most twice. Use lots of tape.)

Mostriamo come una TM a sola scrittura B sia simulabile con una TM deterministica a nastro singolo S.
S memorizza il contenuto di B all'interno di un singolo nastro memorizzando con un simbolo extra la singola scrittura che va a realizzare. Ciò realizza due scritture: la prima per registrare il simbolo da scrivere a la seconda per indicare che è stato copiato, magari con un simbolo con un pallino.
La TM a nastro singolo S funziona come segue:

1) Sostituisce w con la configurazione iniziale del nastro
2) Copia secondo la funzione di transizione di B tutto sul nastro e segue i movimenti a destra e a sinistra di questo

3) Una volta che individua il punto in cui scrivere, aggiorna il nastro scrivendo e, una volta scorso tutto, aggiunge il simbolo delimitatore per indicare di aver raggiunto la fine del nastro
4) Se la macchina raggiunge uno stato di accettazione allora *accetta*, altrimenti *rifiuta*

3.11 A ***Turing machine with doubly infinite tape*** is similar to an ordinary Turing machine, but its tape is infinite to the left as well as to the right. The tape is initially filled with blanks except for the portion that contains the input. Computation is defined as usual except that the head never encounters an end to the tape as it moves leftward. Show that this type of Turing machine recognizes the class of Turing-recognizable languages.

Per risolvere l'esercizio dobbiamo mostrare che:

a) Ogni linguaggio riconosciuto da una TM a doppio nastro infinito è Turing-riconoscibile
b) Ogni linguaggio Turing-riconoscibile dell'alfabeto di TM I è riconosciuto dalla TM a doppio nastro infinito I

a) Il linguaggio di I è Turing-riconoscibile: infatti, considerato un generico alfabeto dato in pasto alla macchina, essa continuerà a scorrere indefinitamente a destra o a sinistra fintanto che non esisterà un modo per farla accettare; essendoci solo una porzione di input, la TM lo scorre tutto andando poi all'estrema sinistra del nastro ad inserire un simbolo, dopo averlo letto tutto, che indica la porzione letta (e che indica la prima lettura); essendo infinito anche a destra, userà dei mark per indicare la fine della lettura e rimpiazza i blank qualora si tratti di fare una scrittura, potenzialmente andando anche in loop. Ad ogni modo, l'infinità del nastro assicura che l'input venga scansionato a prescindere dalla dimensione e di non "scivolare" fuori dalla sua dimensione. In qualsiasi caso, il linguaggio viene riconosciuto.
   Quindi simuliamo M (normale) con I (doppio nastro infinito):
   1) sostituisce la stringa di input con una codifica corrispondente alla macchina di riferimento
   2) segue la funzione di transizione di I
   3) dato che la macchina I ha un doppio nastro infinito, semplicemente, porremo scorrendo tutto l'input un simbolo delimitatore a sinistra, per fare in modo la parte destra del nastro sia come d'abitudine
   4) se raggiunge uno stato d'accettazione, *accetta*, altrimenti riprova dal punto 2

b) Ora simuliamo I con M. Avremo bisogno di una macchina M a doppio nastro finito.
   La simulazione esegue questi passi:
   1) Contiene in un nastro tutta la porzione degli input, che viene copiato nel secondo nastro inizialmente contenente tutti blank
   2) A seconda della lunghezza dell'input esegue la copia della sola stringa w in tutta la sua lunghezza lasciando i blank intorno alla stringa
   3) Una volta coperto tutto l'input, se raggiunge uno stato di accettazione, *accetta*, altrimenti va in loop

oppure

Per dimostrare che questa macchina a doppio nastro infinito sia simile ad una Turing Machine ordinaria, allora deve valere che la macchina ordinaria M sia simulabile dalla macchina I e viceversa.

1) *Simulazione di M per I*

Essendo questa macchina finita, allora basterà marcare la fine a sinistra per D e poi semplicemente continuare a muoversi a destra, prevenendo che la testina si muova a sinistra.

2) *Simulazione di I per M*

Qua dobbiamo simulare la situazione del nastro finito da una parte con la macchina infinita da entrambi i lati. Quindi immaginiamo una macchina a 2 nastri dove, partendo dalla stringa di input:

5) da una parte avremo la fine del nastro e ciò si simula con una serie di simboli blank
6) dall'altra avremo tutto l'input della macchina, ottenuto con i corrispettivi input presenti

Perciò la macchina è equivalente alla TM ordinaria.

Scritto da Gabriel

Answer: A TM with doubly infinite tape can simulate an ordinary TM. It marks the left-hand end of the input to detect and prevent the head from moving off of that end.

To simulate the doubly infinite tape TM by an ordinary TM, we show how to simulate it with a 2-tape TM, which was already shown to be equivalent in power to an ordinary TM. The first tape of the 2-tape TM is written with the input string, and the second tape is blank. We cut the tape of the doubly infinite tape TM into two parts, at the starting cell of the input string. The portion with the input string and all the blank spaces to its right appears on the first tape of the 2-tape TM. The portion to the left of the input string appears on the second tape, in reverse order.

3.12 A ***Turing machine with left reset*** is similar to an ordinary Turing machine, but the transition function has the form

$$\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{R}, \mathrm{RESET}\}.$$

If $\delta(q, a) = (r, b, \mathrm{RESET})$, when the machine is in state q reading an a, the machine's head jumps to the left-hand end of the tape after it writes b on the tape and enters state r. Note that these machines do not have the usual ability to move the head one symbol left. Show that Turing machines with left reset recognize the class of Turing-recognizable languages.

Dobbiamo mostrare che per riconoscere la classe dei linguaggi Turing-riconoscibili, il linguaggio deve essere nella forma:

$$L = L(M) = \{w \in \Sigma^* : q_0 w \overset{*}{\Rightarrow} u q_{\text{accept}} v, \text{ for some } u, v \in \Gamma^* \}$$

cioè aggiungendo la transizione di *left-reset*, si riconosce lo stesso linguaggio.
In particolare, la macchina TM M è ordinaria, mentre M' implementa left-reset; quando M si muove a destra, M' farà lo stesso. Sappiamo invece che la macchina M' qualora si muova a sinistra, sposta il contenuto del proprio nastro una posizione più a destra per ogni simbolo. Dato che deve essere riconoscibile da una TM normale, quando M si sposta a sinistra, M' marca il simbolo con un mark, per ricordare di dove rispostare il nastro, facendo un nuovo reset per riportare tutti i simboli allo stesso stato di M, così entrambi hanno fatto la stessa cosa.
Idea completa:

Let M be an ordinary TM; let M' be the "left-reset" machine that simulates M as follows: if M makes a right-transition, M' does the same. If M makes a left-transition, M' acts as follows: it marks the current position of the head. To place this mark, if the current square contains a $b \in \Gamma$, M' replaces it with $\hat{b}$. The understanding is that $\Gamma_{M'} = \Gamma_M \cup \hat{\Gamma}_M$, where $\hat{\Gamma}$ consists of the "hatted" versions of all the symbols of Γ. Then M' does a left-reset, and moves the contents of each square one position to the right, except for the hatted square, i.e., the "hat" remains in the same tape position. Once all the squares have been shifted one position to the right, M' does a second left-reset, and travels with right-transitions until it reaches the hatted square, where it now does whatever M would have done.

Facendolo alla Breso:

1) Simuliamo il comportamento della TM con reset a sinistra R con una equivalente standard M.
   Essa ragiona in base alla mossa da eseguire, che può prevedere un eventuale reset, quindi la testina "salta" a sinistra. Quando ciò avviene, ha bisogno di un simbolo per ricordarsi dove saltare esattamente e deve essere un punto oltre la posizione iniziale dell'input, in particolare in un punto precisamente indicato.
   La simulazione di M, dunque, è composta da:

- la scrittura di un particolare simbolo che faccia ricordare il punto prima dell'input
- dipende tutto dalla funzione di transizione, in quanto se ci dobbiamo muovere a destra, scriviamo il simbolo sul nastro e si va verso lo stato r
- se ci dobbiamo muovere a sinistra, eseguiamo la transizione reset, scrivendo il simbolo sul nastro, muovendo la testina a sinistra e andando nello stato di reset. Quando ciò avviene, la testina si sposta ancora a sinistra di una cella e poi ritorna nel vecchio simbolo iniziale
- se arriva ad uno stato di accettazione o di rifiuto, allora esegue una delle due cose, altrimenti torna al secondo passaggio

2) Simuliamo il comportamento della TM standard M con una a reset a sinistra R.
In questo caso abbiamo la transizione di reset qualora ci si debba muovere a sinistra ma preservando il comportamento standard.
La simulazione si compone di:
1) Scrivere il simbolo iniziale sul nastro e porre, alla fine della ricopiatura dell'input, un particolare simbolo a scelta, mettiamo *
2) A seconda della funzione di transizione, se mi devo muovere a sinistra, allora la macchina esegue la mossa di reset; dato però che siamo in una macchina con computazione finita, essa deve andare oltre l'ultimo reset, simulando l'andamento a sinistra. Nel qual caso, marca l'ultimo reset avvenuto una volta compiuto e ci spostiamo gradualmente a sinistra, scrivendo i simboli sul nastro e andando verso lo stato *reset* successivo; se non fossero stati marcati reset, avrebbe eseguito comunque la mossa e lo resetta nuovamente
3) Se ci muoviamo a destra, allora la macchina simula l'andamento a destra, scrivendo il simbolo sul nastro e spostandosi allo stato *r*
4) Se non si è nello stato di accettazione o rifiuto, si torna al secondo passo

3.13 A ***Turing machine with stay put instead of left*** is similar to an ordinary Turing machine, but the transition function has the form

$$\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{R, S\}.$$

At each point, the machine can move its head right or let it stay in the same position. Show that this Turing machine variant is *not* equivalent to the usual version. What class of languages do these machines recognize?

Il linguaggio è Turing-riconoscibile, perché ad un certo punto la computazione potrebbe andare in loop (dato che al posto di andare a sx si va in stay). Dobbiamo dunque mostrare che esiste una TM che riconosce questo linguaggio, ma potrebbe essere sottoposto a loop.
Semplicemente si usa una TM M che ha le stesse transizioni e continua la sua computazione leggendo a destra fino alla fine dell'input; nel qual caso accetta. Si muove solo a destra come un DFA e ogniqualvolta si ha una transizione *stay*, il nastro simula l'idea del blank, ma estende sempre un nuovo simbolo alla funzione di transizione, in maniera tale che continui a computare il linguaggio.
La funzione di transizione è descritta da:

$$\delta'(q,\sigma) = \begin{cases} q, & \text{if } q \in \{q_{\texttt{accept}}, q_{\texttt{reject}}\} \\ q_{\texttt{reject}}, & \text{if } M \text{ starting at state } q \text{ and reading } \sigma \text{ keeps staying put.} \\ q', & \text{where } q' \text{ is the state the } M \text{ enters when it first moves right} \\ & \quad \text{upon starting at state } q \text{ and reading } \sigma. \end{cases}$$

Siccome potenzialmente potrebbe sempre andare in loop, non è come le altre macchine Turing-riconoscibili; o va in loop e sta in fase stay oppure va avanti.

2. (**Eco-friendly TM**) An *eco-friendly* Turing machine (ETM) is the same as an ordinary (deterministic) one-tape Turing machine, but it can read and write on both sides of each tape square: front and back.

   At the end of each computation step, the head of the eco-friendly TM can move left (L), move right (R), or flip to the other side of the tape (F).

Show that eco-friendly TMs recognize the class of Turing-recognizable languages. That is, use a simulation argument to show that they have exactly the same power as ordinary TMs.

Per mostrare che una eco-friendly TM riconosce la classe dei linguaggi Turing-riconoscibili, dobbiamo dimostrare:

1) Il linguaggio di una TM Eco-friendly è Turing-riconoscibile
2) Una TM a nastro singolo simula correttamente TM Eco-friendly

1) Descriviamo una TM S a nastro singolo che descrive il comportamento di una TM eco-friendly. Dato che la testina può spostarsi liberamente a sinistra, destra e da tutt'altro lato del nastro, deve tenere traccia delle due corrispondenti posizioni più a sinistra e più destra del nastro, marcando le due corrispondenti posizioni. Seguendo la funzione di transizione di M la macchina si sposta.
   S = "Su input w1, w2, … wn":
   a. Sostituisce la configurazione iniziale del nastro con i simboli corrispondenti della TM eco-friendly
   b. Scorre il nastro dall'inizio dell'input (individuando una cella blank prima di esso) e marca il simbolo più a sinistra e scorre tutto l'input arrivando alla parte più a destra, sempre marcando la cella più a destra
   c. A seconda della funzione di transizione, ragiona in questo modo:
      1) Se δ(r, a) = (s, b, L), la macchina scrive b non marcato sulla cella corrente, si sposta a sinistra e va verso lo stato L
      2) Se δ(r, a) = (s, b, R), la macchina scrive b non marcato sulla cella corrente, si sposta a sinistra e va verso lo stato R
      3) Se δ(r, a) = (s, b, F), la macchina scrive b non marcato sulla cella corrente, poi la macchina si sposta a sinistra cercando il simbolo marcato. Essendo che si deve muovere a destra o a sinistra "saltandovi direttamente", allora poco importa se in questa transizione, la macchina vada a destra o sinistra. La macchina, una volta trovato il simbolo più a sinistra marcato, va nello stato F spostando la testina dall'altra cella marcata dall'altra parte scorrendo tutto il nastro a destra.
      4) Se non siamo nello stato di accettazione o rifiuto, riprendi dal passo 2

2) Ora convertiamo una TM standard in una TM Eco-friendly. Come prima, se devo spostarmi a sinistra o destra, poco cambia, la cosa importante è descrivere il funzionamento per eseguire lo shift dall'altra parte.
   M = "Su input w: "
   a. Prima dell'input, controlla che il nastro sia blank e pone sulla prima stringa dell'input un mark per ricordarsi della transizione. Scorre tutto l'input e pone il mark alla fine di questo.
   b. Simula il comportamento di S, infatti se la mossa da simulare è δ(r, a) = (s, b, L) si sposta a sinistra scrivendo b sul nastro e va allo stato s, se δ(r, a) = (s, b, R), si sposta a destra scrivendo b sul nastro e va allo stato s, mentre per la transizione F, scrive b sul nastro e si sposta a sinistra fino a trovare il primo simbolo più a sinistra con un mark. Una volta trovato, lo pone a blank e pone il mark sul simbolo a destra di quest'ultimo che dovrà essere non vuoto. Se ciò accade, l'input esiste ancora. A questo punto saltiamo alla parte destra del nastro e qui similmente, spostiamo il mark a sinistra e poniamo blank sulla cella attuale. La cosa continua ricorsivamente finché non esaurisce tutta la computazione.
   c. Se non siamo nello stato di accettazione o rifiuto, riprendi dal passo 2

Esercizi Linguaggi decidibili/Turing-riconoscibili/Linguaggi indecidibili

[A]**4.10** Let $INFINITE_{\text{DFA}} = \{\langle A \rangle |\ A$ is a DFA and $L(A)$ is an infinite language$\}$. Show that $INFINITE_{\text{DFA}}$ is decidable.

Assumendo di avere un DFA che accetta il linguaggio sull'input di A, a seconda del numero di stringhe "n" dello stesso, l'automa da costruire accetterà un numero stringhe almeno pari ad A e poi continuerà ad andare idealmente avanti nella computazione.
L'idea è che la macchina TM M:

1) abbia almeno un numero "n" di stati
2) costruisca un automa DFA B tale da accettare tutto l'insieme "n" delle stringhe
3) costruisca un automa DFA C tale che esso sia intersezione di A e di B (operazione chiusa), una volta scorso sul nastro tutto A, segni la sua fine e con B non fa altro che continua a scorrere il nastro indefinitamente
4) dato che deve essere infinito, se B accetta allora *rifiuta*, altrimenti *accetta*

Essendo un automa deterministico, si sa che ad un certo punto della computazione A ha una fine ottenuta in maniera deterministica; valenti le condizioni spiegate allora B continuerà a computare avanzando.
Poste queste condizioni, il linguaggio può dirsi *decidibile.*

4.11 Let $INFINITE_{\text{PDA}} = \{\langle M \rangle |\ M$ is a PDA and $L(M)$ is an infinite language$\}$. Show that $INFINITE_{\text{PDA}}$ is decidable.

4.11)
In questo caso abbiamo un PDA per A quindi come tale deve accettare e scorrere nella pila fino idealmente a svuotarsi una CFG. Assumiamo che tale grammatica sia in forma normale di Chomsky; se questo avviene, andremo ad un certo punto incontro a regole terminali.
Quindi idealmente con una TM:

1) iniziamo dalla prima regola e marchiamo in successione ogni altra
2) essendo in formale normale di Chomsky, si può usare il pumping Lemma ed immaginare ci sia sempre una possibile derivazione aVb per una variabile V non vuota
3) per Chomsky esisterà sempre una derivazione che permette di ottenere V → *aBb, avendo quindi una derivazione ciclica

Volendolo vedere con un grafo:

1) si genera un grafo tale che per ogni derivazione della grammatica vi siano almeno due vertici connessi
2) si continua procedendo sequenzialmente nel nastro per ogni pezzo connesso
3) se si verifica un ciclo prima della fine di ogni regola, rifiuta, altrimenti ha percorso tutte le regole3 ed accetta

Poste queste condizioni, il linguaggio può dirsi *decidibile.*

4.13 Let $A = \{\langle R, S \rangle |\ R$ and S are regular expressions and $L(R) \subseteq L(S)\}$. Show that A is decidable.

Sono entrambe due espressioni regolari tali che una contenga l'altra come sottoinsieme.
A livello pratico significa che una porzione di linguaggio di R è contenuta in S; quindi, entrambe le espressioni hanno almeno una parte dello stesso linguaggio.
Si può quindi idealmente ragionare con una TM N che decide S_{REX} in questo modo:

- trasforma R ed S in due ε-NFA equivalenti B e C
- si esegue N sugli input di R ed S
- se scorrendo tutto il linguaggio di R si arriva, marcando tutti i suoi stati, a coprire almeno una parte di stati di S, anche essi marcati allora *accetta*, altrimenti *rifiuta*

Conviene quindi avere N multinastro, tale da avere gli input da una parte, controllando R su un nastro e vedere sul risultante terzo nastro dove starebbe S se vi è un match del pattern precedente.

Poste queste condizioni, il linguaggio può dirsi *decidibile*.

Say that an NFA is ***ambiguous*** if it accepts some string along two different computation branches. Let $AMBIG_{\text{NFA}} = \{\langle N \rangle |\ N \text{ is an ambiguous NFA}\}$. Show that $AMBIG_{\text{NFA}}$ is decidable. (Suggestion: One elegant way to solve this problem is to construct a suitable DFA and then run E_{DFA} on it.)

Sappiamo che l'NFA computa in maniera non deterministica i propri input; pertanto, dobbiamo dimostrare che accetti le tutte le computazioni. Sfruttiamo il suggerimento dell'esercizio e:
Si può quindi idealmente ragionare con una TM N che decide $AMBIG_{NFA}$ in questo modo:

- trasforma N da un NFA ad un DFA equivalente tramite riconversione
- si esegue N sull'input, ora in maniera deterministica, scorrendo tutte le stringhe del linguaggio sapendo che lo stato iniziale sarà lo stesso ma, essendo una grammatica ambigua, avrà più stati a destra e a sinistra. Tutto dipende dall'input accettato dall'NFA; per ogni stringa accettata, si avranno due possibili stringhe.
- si marcano tutte le possibili stringhe corrispondenti alle posizioni dell'NFA ed occorrono almeno due simboli per sapere se la computazione è nello stato attuale e quali stringhe siano state percorse. Per più stringhe accettiamo infatti lo stesso linguaggio.
- mettendo un simbolo sullo stato iniziale, marchiamo due colori per indicare se abbiamo fatto una mossa o un'operazione di inserimento/cancellazione, spostandoci in avanti. Gli stati d'accettazione avranno idealmente lo stesso simbolo oppure due simboli di accettazione diversi.
- Se abbiamo più simboli di accettazione una volta raggiunti, nel qual caso il DFA ha percorso deterministicamente le varie strade possibili di NFA, accettando
- se rimane un unico simbolo durante tutta la computazione, comunque accetta, avendo avuto un simbolo iniziale, un mark per i simboli raggiunti e il simbolo finale corrispondente, abbiamo percorso ritogliendo tutti i mark le posizioni dell'NFA e accettiamo come si vuole

Claim 1. The following language is undecidable.

$$\text{TOTAL} = \{\langle M \rangle \mid M \text{ is a TM that halts on all inputs}\}$$

Dobbiamo ragionare per contraddizione ed assumere TOTAL sia decidibile.
Affermiamo che esiste una TM che riconosce il linguaggio complementare A:
ATM = {<M, w> | è una TM che accetta w}
La TM N su input <M, w> dove M è una TM:

1) Costruisce la TM 1M definita come:
   M = Su input x
   a. Esegue M su x
   b. Se viene riconosciuto accetta, altrimenti rifiuta o va in loop
2) Usa l'output <M, w> per poter eseguire M su W
3) se M accetta, *accetta*
4) se M rifiuta o va in loop, rifiuta

Abbiamo appena dimostrato che esiste un decisore, prendendo <M, w>, sapendo che M accetta w e quindi la M complementare accetterà tutte le stringhe. Quindi N accetta <M, w>, avendo un decisore. Contrariamente, siccome M rifiuta se non trova la stringa w, allora M rifiuterà tutte le altre stringhe. Pertanto, M_{TOTAL} rifiuta M* (tutte le stringhe) e accetta solo l'input <M, w>; tuttavia sappiamo che il linguaggio è indecidibile e non può esistere per questo motivo un decisore.

$$S = \{\langle M \rangle \mid M \text{ is a TM that accepts } w \text{ if and only if it accepts } w^R\}$$

Partiamo da una funzione di riduzione, che ragioni come prima su una macchina TM M.
Questa è una variante del problema sotto.
Infatti, la TM M su input w:

- comincia a scorrere il nastro, cercando di marcare tutto *w*
- se ciò non accade *rifiuta*, altrimenti avanza la computazione

- a questo punto, alla fine del nastro ci sarà la stringa palindroma e, se completamente riconosciuta, *accetta*
- se a priori l'input è diverso rifiuta

La funzione di riduzione f su <M, w>:

- esegue M su w
- se M accetta, *accetta*, altrimenti *rifiuta*

Se avessimo un input del tipo {01, 10} accetta e M ∈ S; similmente se avessimo un input del tipo {01} allora M non appartiene ad S. La funzione è computabile e abbiamo dimostrato che esiste un decisore; tuttavia il linguaggio è indecidibile e ciò non ha senso.

Exercise 11.4 (Undecidable Grammar Problems, 1.5+1.5 Points)

The *equivalence problem* and the *intersection problem* for general grammars are defined as:

- EQUIVALENCE: Given two general grammars G_1 and G_2, is $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?
- INTERSECTION: Given two general grammars G_1 and G_2, is $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$?

(a) Show that EQUIVALENCE is undecidable, by reducing EMPTINESS to it.

(b) Show that INTERSECTION is undecidable, by reducing EMPTINESS to it.

L'idea è di sfruttare l'idea del linguaggio vuoto, dunque dicendo prima che:

1) date due grammatiche, se sono uguali accettiamo.
   Per descrivere questo problema, usiamo l'idea del linguaggio vuoto, questa volta avendo M_1 ed M_2 che sono due TM costruite su una stringa w con la seguente funzione di riduzione *f*.
   f = Su input <M, w> dove M è una TM e w è una stringa:
   1) Esegue M_1 su w
   - avanza sul nastro e se riconosce tutta *w*, *accetta*
   - altrimenti rifiuta
   2) Esegue M_2 su w
   - avanza sul nastro e se riconosce tutta *w*, *accetta*
   - altrimenti rifiuta
   3) Dà in output <M_1, M_2>

   Le due macchine accettano solo se accettano entrambe; passando alla riduzione, avremmo che la funzione M_1, M_2 ragiona con il complemento del test del vuoto, il quale accetterà esattamente tutte le stringhe che *non* fanno parte della grammatica. Tuttavia, vogliamo mostrare che L sia indecidibile, dunque tale decisore non può esistere.
2) date due grammatiche, se almeno una delle due riconosce tutte le stringhe in Σ^* e l'altra che si ferma su tutti gli input. Ciò sarebbe la soluzione all'*halting problem*.
   Infatti, avremmo una funzione f che esegue su <M_1, M_2, w>:
- simula M_1 sul nastro fermandosi per ogni singolo simbolo che fa parte di *w*
- similmente, simuliamo M_2 sul nastro, ma questa dovrà rifiutare tutte le stringhe, fermandosi.
  Siccome entrambe queste macchine accettano il rispettivo linguaggio, abbiamo ridotto al problema del vuoto, ma sappiamo che un decisore normalmente non esiste. Quindi il linguaggio è indecidibile.

Consider the problem of determining whether a two-tape Turing machine ever writes a non-blank symbol on its second tape, i.e.

$$N = \{\langle M, w\rangle \mid M \text{ is a two-tape Turing machine which writes a non-blank symbol onto its second tape when it runs on } w\}.$$

Show that N is undecidable. *Hint*: Use a reduction from A_{TM}.

Assumiamo che N sia decidibile e cerchiamo di costruire una funzione f di riduzione tale che <M, w> sia riconosciuto.

Partiamo proprio dalla macchina M:

- questa è una macchina a due nastri che semplicemente scorre *w* ed ogni volta che marca una stringa sul primo nastro, scrive un simbolo non blank sul secondo; il simbolo dovrà essere garantito nella funzione di transizione e correttamente anche mappare il caso dei simboli vuoti, che dovrà essere definito.
- se la macchina eseguendo scrive almeno un simbolo non blank sul nastro accetta, altrimenti rifiuta

A questo punto avremo:

f <M, w> che su input M che è una TM e w che è una stringa:

- *accetta*, se M accetta w

- *rifiuta*, se M rifiuta *w*

Abbiamo dunque dimostrato che esiste un decisore, ma il linguaggio è indecidibile e il decisore non può esistere.

Da appello 05/07/2021

2. Dimostra che il seguente linguaggio è indecidibile:

$$L_2 = \{\langle M, w\rangle \mid M \text{ accetta la stringa } ww^R\}.$$

Partiamo dal costruire una TM M che effettivamente riconosca la stringa ww^R, quindi la stringa che abbiamo e similmente la stringa palindroma. Per fare ciò immaginiamo che:
M = Su input w:

- si ha il primo carattere della prima stringa, lo esaminiamo. Se appartiene alla stringa w, prosegue la computazione e si muove alla fine del nastro; altrimenti, rifiuta
- da questa parte si ha la stringa palindroma, dunque si esamina se tale carattere appartiene al palindromo, se così non fosse rifiuta.
- continuando a muoversi su e giù nella testina, la macchina *accetta* se esaurisce in questo modo tutta la computazione, *rifiuta* altrimenti.

Abbiamo quindi creato un decisore. Ora creiamo una funzione di riduzione f, che prende in input la TM M appena creata ed una stringa *w*, dimostrando quindi che $A_{TM} \leq_m L_2$ e se e solo se f<M, w> = <M> ∈ L_2

La funzione di decisione f

- esegue M sull'input w
- se M accetta, *accetta*
- se M rifiuta, allora *rifiuta*

Se l'input appartiene correttamente al linguaggio, la TM rifiuta. La funzione calcola correttamente tutti gli input e si ferma se avesse una cosa del tipo (01), rispetto ad una stringa accettabile come (0110). Siccome f è computabile in un numero finito di passi, si fermerà su tutti gli input ed f rappresenta una riduzione del linguaggio, rifiutando tutte le stringhe che non centrano.
Tuttavia, sappiamo che L_2 è indecidibile ed è una contraddizione, dunque non può esistere un decisore.

5.32 Prove that the following two languages are undecidable.

a. $OVERLAP_{CFG} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs where } L(G) \cap L(H) \neq \emptyset\}$.
(Hint: Adapt the hint in Problem 5.21.)

b. $PREFIX\text{-}FREE_{CFG} = \{\langle G\rangle \mid G \text{ is a CFG where } L(G) \text{ is prefix-free}\}$.

a) Definiamo le due grammatiche con G costituita da un insieme di simboli $t_i\, a_i$ fino a $t_m\, a_m$
Similmente, definiamo H costituita da un insieme di simboli $b_j\, a_j$ fino a $b_n\, a_n$
(questo perché ci deve stare un'intersezione).
Se entrambi sono non vuoti:

- la prima TM descrive la prima CFG, rifiuta se non è in quella forma
- la seconda TM descrive la seconda CFG, rifiutando se non è in quella forma.
  Se la grammatica corrispondesse, otterremmo: $t_i\, a_i \ldots t_m\, a_m$ fino a $b_j\, a_j \ldots b_n\, a_n$
  Usiamo una funzione di riduzione che prendendo le due TM precedenti:
- toglie tutte le regole terminali; se corrisponde alla stringa ottenuta alla fine della precedente, *accetta*, altrimenti *rifiuta*.
  Abbiamo ragionato per assurdo, affermando l'esistenza del decisore, in realtà non esistente.
  Dunque, L non è decidibile.

b) Definiamo prima G come una CFG, immaginando una grammatica che rispetti entrambe le condizioni:

- deve essere CFG (possibilmente in forma di Chomsky)

- deve essere prefix-free

Usiamo quindi una CFG che ha come regole sulle stringhe <w, x>

S → A | B
A → wAx | wx
B → xBw | xw

Generalizzando in Chomsky, avremmo l'aggiunta delle regole terminali per wx e xw (aggiungendo ad esempio C e D appunto per wx e xw, ma poco cambia ai fini dell'esercizio), ma sappiamo che G è CFG.
Ora, per discutere il discorso dell'essere liberi da prefissi, dobbiamo presupporre che le derivazioni con le stringhe vuote portino ad avere le stesse stringhe.
Idealmente quindi avendo un ε sulle singole regole, applicando il principio della leftmost derivation, tutti i simboli devono matchare. Sapendo che Chomsky elimina le regole vuote, ciò dovrà essere vero.
Significa quindi che A → a' | ε e B' → b' | ε allora avremo A → a' e B → b'.
Se sono prefix-free, allora avremo due TM che banalmente scorrono tutte le derivazioni sul nastro e *accettano* se le due computazioni, togliendo le stringhe vuote, corrispondono.

La funzione di riduzione su <M, w> allora:

- M su input w scansione tutti i prefissi di y su x
- poi *rifiuta*
- esegue M su x
- se x è accettata da M, allora *accetta*
- se x è rifiutata da M, allora *rifiuta*

In questo modo, rifiutando tutte le stringhe prefisse, togliamo come descritto prima le regole terminali,
Il decisore però non deve esistere, dato che L è indecidibile.

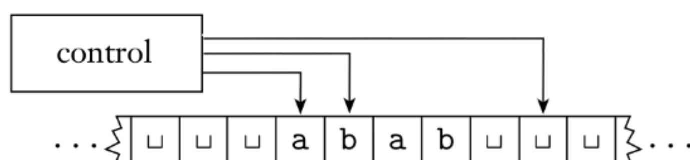
Una macchina di Turing a testine multiple è una macchina di Turing con un solo nastro ma con varie testine. Inizialmente, tutte le testine si trovano sopra alla prima cella dell'input. La funzione di transizione viene modificata per consentire la lettura, la scrittura e lo spostamento delle testine. Formalmente,

$$\delta : Q \times \Gamma^k \mapsto Q \times \Gamma^k \times \{L, R\}^k$$

dove k è il numero delle testine. L'espressione

$$\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, L, R, \dots, L)$$

significa che, se la macchina si trova nello stato q_i e le testine da 1 a k leggono i simboli $a_1, \dots, a_k$ allora la macchina va nello stato q_j, scrive i simboli $b_1, \dots, b_k$ e muove le testine a destra e a sinistra come specificato.



Un esempio di TM con tre testine.

Mostriamo come simulare una TM a testine multiple B con una TM deterministica a nastro singolo S.
Essendo una macchina che riporta molteplici testine su un nastro singolo, ciascuna di queste testine deve essere memorizzata in qualche modo posizionalmente parlando, per esempio assumiamo di separarle per mezzo di un pallino •, capendo dove è stata l'ultima scrittura, si sposta in avanti marcando la scrittura scrivendo l'insieme di simboli di b e prosegue in questo modo.
Ciascuna sequenza di stringhe memorizzata è separata da un # e tramite le marcature siamo in grado di capire dove e quando si sono avute scritture particolari.
La TM S a nastro singolo funziona come segue:

- si parte dalla prima cella e, avendo un certo numero di simboli da leggere, una delle testine si muove e non appena finisce la sua lettura, marca con un simbolo • la posizione
- le successive testine cominciano la lettura nello stesso modo, sapendo che le rispettive porzioni sono separate da # e quindi si ha la scansione di ciascuna porzione. In particolare, ogni testina, comincerà a riscrivere partendo da •
- la scrittura del nastro avviene secondo la funzione di transizione di B:
  - se δ(r,a) = (s,b,L), scrive *b* non marcato sulla cella corrente, dopodiché si sposta immediatamente a sinistra e sposta di una cella a sinistra tutte le marcature con il pallino, fino a che non raggiunge il #, al posto del quale aggiungerà un blank marcato, indicando che ha compiuto la sua scrittura a sinistra.
  - se δ(r,a) = (s,b,R) scrive *b* non marcato sulla cella corrente, dopodiché si sposta immediatamente a destra e sposta di una cella a destra tutte le marcature con il pallino, fino a che non raggiunge il #, al posto del quale aggiungerà un blank marcato, indicando che ha compiuto la sua scrittura a destra.

Se questa simulazione raggiunge lo stato di accettazione di B allora accetta, altrimenti se raggiunge lo stato di rifiuto di B, allora rifiuta. Se non dovesse essere nessuno dei due, si ripeterebbe la simulazione partendo dallo scorrimento del nastro.

Una soluzione alternativa:

<u>Solution</u> We must prove that the k head TM can be simulated by an ordinary TM. Since we already know that a multiple-tape TM can be simulated with an ordinary one, it suffices to simulate the k-head TM with a multiple-tape TM. We call K to the k-tape TM and M to the multiple-tape TM that simulates the former.

Assume we enumerate each head as h_i with $i = 1, \dots, k$. We use a $k+1$-tape TM M to do the simulation. The first step is to copy the input tape of K into each tape of M. In the last tape, we call it 'extra', we add a mark to each symbol where there is a head in K (add a mark means that the alphabet of machine M is the union of the alphabet of the machine K plus the set of all the symbols in the alphabet, with a mark).

So, we get the following picture:

Let the machine K be in the following configuration:

	$\downarrow_1$		$\downarrow_{23}$				$\downarrow_4$				
	σ	α	β	δ	α	σ	β	β	σ	α	α

Then the machine M is in the following configuration:

	$\downarrow$										
	σ	α	β	δ	α	σ	β	β	σ	α	α
			$\downarrow$								
	σ	α	β	δ	α	σ	β	β	σ	α	α
			$\downarrow$								
	σ	α	β	δ	α	σ	β	β	σ	α	α
							$\downarrow$				
	σ	α	β	δ	α	σ	β	β	σ	α	α
$\downarrow$											
	$\hat{\sigma}$	α	$\hat{\beta}$	δ	α	σ	$\hat{\beta}$	β	σ	α	α

At each move in machine K, we do the same movement in the corresponding tape. Then, we check the last (extra) tape to see what are the changes with respect to each tape. We synchronize those changes one by one in the extra tape. For example, if in the tape 2th the 3th cell has been changed from β to σ and the head moves to the right, then after synchronizing such tape, the extra tape will looks as follows

	$\hat{\sigma}$	α	σ	$\hat{\delta}$	α	σ	$\hat{\beta}$	β	σ	α	α

Then when we have to synchronize the 3th tape we compare all the cells except those already synchronized (all except the 3th cell in our example), so the extra tape remain unchanged.

After having synchronized all the tapes. We replicate the extra tape in all the other tapes, which finish the synchronization process and the movement.

Fornisci una descrizione a livello implementativo di una TM deterministica a nastro singolo che decide il linguaggio

$$L_3 = \{u\#w_1\#\ldots\#w_n \mid u, w_i \in \{0,1\}^* \text{ ed esiste } w_j \text{ tale che } u = w_j\}$$

$M =$ "Su input $u\#w_1\#\ldots\#w_n$:

La macchina esegue questi passi:

- marca il simbolo più a sinistra del nastro e controlla che oltre ad esso ci sia solo un simbolo blank; se ciò non avviene rifiuta
- partendo da *u*, marca ogni pezzo di nastro che può contenere il match sul # e verifica che una stringa faccia match; se *w* e *u* sono diverse allora marca il #. Se non esistono # marcati, vuol dire che tutte le stringhe della possibile computazione sono diverse e rifiuta
- continua a zig-zag tra le porzioni delimitate dai cancelletti; se ne trova una uguale *accetta*, altrimenti riprende dal punto 2

Fornisci una descrizione a livello implementativo di una TM deterministica a nastro singolo che decide il linguaggio

$$L_3 = \{ww \mid w \in \{0,1\}^*\}$$

$M =$ "su input x, dove x è una stringa:

- La macchina controlla che la stringa sia composta esattamente dagli stessi pezzi in due parti; pertanto a metà nastro pone un simbolo # e sposta tutti i simboli a destra, avendo così un blank in più prima del primo simbolo
- Esamina la prima metà di nastro e marca il primo simbolo; procede oltre il primo simbolo non marcato dopo il # e verifica che sia lo stesso simbolo; se non è uguale già rifiuta
- Continua esaminando tutte le stringhe fintanto che a sx non abbiamo un simbolo blank; se sono tutte uguali le stringhe fino a quel momento, *accetta*, altrimenti rifiuta

Oppure:

1. Scorre la stringa x per controllare se il numero di simboli è pari o dispari. Se è dispari, rifiuta.
2. Divide la stringa x in due parti uguali. Per farlo marca il primo simbolo di x con un pallino, poi va alla fine della stringa e marca l'ultimo simbolo con un pallino. Continua a marcare un carattere all'inizio e uno alla fine della stringa muovendosi a zig zag.
3. L'ultimo carattere marcato è la posizione di inizio della seconda metà della stringa. Toglie tutte le altre marcature e ritorna all'inizio della stringa.
4. sostituisce il simbolo all'inizio della stringa con $\#$, poi procede a destra e controlla se il simbolo marcato con un pallino è uguale al primo simbolo. Se sono diversi rifiuta.
5. Se i due simboli sono uguali, sostiuisce il simbolo marcato con $\#$ e sposta la marcatura al simbolo successivo.
6. Torna all'inizio della stringa: se ci sono ancora simboli diversi da $\#$, ripeti da 4, altrimenti accetta."

Una macchina di Turing bidimensionale *utilizza una griglia bidimensionale infinita di celle come nastro. Ad ogni transizione, la testina può spostarsi dalla cella corrente ad una qualsiasi delle quattro celle adiacenti. La funzione di transizione di tale macchina ha la forma*

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{\uparrow, \downarrow, \rightarrow, \leftarrow\},$$

dove le frecce indicano in quale direzione si muove la testina dopo aver scritto il simbolo sulla cella corrente.

Dimostra che ogni macchina di Turing bidimensionale può essere simulata da una macchina di Turing deterministica a nastro singolo.

Mostriamo come simulare una TM bidimensionale B con una TM deterministica a nastro singolo S. S memorizza il contenuto della griglia bidimensionale sul nastro come una sequenza di stringhe separate da #, ognuna delle quali rappresenta una riga della griglia. Due cancelletti consecutivi ## segnano l'inzio e la fine della rappresentazione della griglia. La posizione della testina di B viene indicata marcando la cella con $\frown$. Nelle altre righe, un pallino $^{\bullet}$ indica che la testina si trova su quella colonna della griglia, ma in una riga diversa. La TM a nastro singolo S funziona come segue:

S = "su input w:

1. Sostituisce w con la configurazione iniziale $\#\#w\#\#$ e marca con $\frown$ il primo simbolo di w.
2. Scorre il nastro finché non trova la cella marcata con $\frown$.
3. Aggiorna il nastro in accordo con la funzione di transizione di B:
   - Se $\delta(r, a) = (s, b, \rightarrow)$, scrive b non marcato sulla cella corrente, sposta $\frown$ sulla cella immediatamente a destra. Poi sposta di una cella a destra tutte le marcature con un pallino. Se in qualsiasi momento S sposta una marcatura sopra un #, S scrive un blank marcato al posto del # e sposta il contenuto del nastro da questa cella fino al ## finale, di una cella più a destra.
   - Se $\delta(r, a) = (s, b, \leftarrow)$, scrive b non marcato sulla cella corrente, sposta $\frown$ sulla cella immediatamente a sinistra. Poi sposta di una cella a sinistra tutte le marcature con un pallino. Se in qualsiasi momento S sposta una marcatura sopra un #, S scrive un blank marcato al posto del # e sposta il contenuto del nastro da questa cella fino al ## iniziale, di una cella più a sinistra.
   - Se $\delta(r, a) = (s, b, \uparrow)$, scrive b marcato con un pallino nella cella corrente, e sposta $\frown$ sulla prima cella marcata con un pallino posta a sinistra della cella corrente. Se questa cella marcata non esiste, aggiunge una nuova riga composta solo da blank all'inizio della configurazione.
   - Se $\delta(r, a) = (s, b, \downarrow)$, scrive b marcato con un pallino nella cella corrente, e sposta $\frown$ sulla prima cella marcata con un pallino posta a destra della cella corrente. Se questa cella non esiste, aggiunge una nuova riga composta solo da blank alla fine della configurazione.

4. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di B, allora *accetta*; se la simulazione raggiunge lo stato di rifiuto di B allora *rifiuta*; altrimenti prosegue con la simulazione dal punto 2."

(8 punti) Una *Turing machine con alfabeto binario* è una macchina di Turing deterministica a singolo nastro dove l'alfabeto di input è $\Sigma = \{0, 1\}$ e l'alfabeto del nastro è $\Gamma = \{0, 1, \sqcup\}$. Questo significa che la macchina può scrivere sul nastro solo i simboli $0, 1$ e blank: non può usare altri simboli né marcare i simboli sul nastro.

Dimostra che che le Turing machine con alfabeto binario machine riconoscono tutti e soli i linguaggi Turing-riconoscibili sull'alfabeto $\{0, 1\}$.

Per risolvere l'esercizio dobbiamo dimostrare che

a) ogni linguaggio riconosciuto da una Turing machine con alfabeto binario è Turing-riconoscibile

b) ogni linguaggio Turing-riconoscibile sull'alfabeto {0, 1} è riconosciuto da una Turing machine con alfabeto binario

Quindi:

a) Questo caso è semplice: una Turing machine con alfabeto binario è un caso speciale di Turing machine deterministica a nastro singolo. Quindi ogni linguaggio riconosciuto da una Turing machine con alfabeto binario è anche Turing-riconoscibile.

b) Per dimostrare questo caso, consideriamo un linguaggio L Turing-riconoscibile, e sia M una Turing machine deterministica a nastro singolo che lo riconosce. Questa TM potrebbe avere un alfabeto del nastro Γ che contiene altri simboli oltre a 0, 1 e blank. Per esempio, potrebbe contenere simboli marcati o separatori. Per costruire una TM con alfabeto binario B che simula il comportamento di M dobbiamo come prima cosa stabilire una codifica binaria dei simboli nell'alfabeto del nastro Γ di M. Questa codifica è una funzione C che assegna ad ogni simbolo a ∈ Γ una sequenza di k cifre binarie, dove k è un valore scelto in modo tale che ad ogni simbolo corrisponda una codifica diversa. Per esempio, se Γ contiene 4 simboli, allora k = 2, perché con 2 bit si rappresentano 4 valori diversi. Se Γ contiene 8 simboli, allora k = 3, e così via.

La TM con alfabeto binario B che simula Me definita in questo modo:

B = Su input w:

1) Sostituisce $w = w_1 w_2 \dots w_n$ con la codifica binaria $C(w_1)C(w_2)\dots C(w_n)$, e riporta la testina sul primo simbolo di $C(w_1)$.
2) Scorre il nastro verso destra per leggere k cifre binarie: in questo modo la macchina stabilisce qual è il simbolo a presente sul nastro di M. Va a sinistra di k celle.
3) Aggiorna il nastro in accordo con la funzione di transizione di M:
   a. Se $\delta(r, a) = (s, b, R)$, scrive la codifica binaria di b sul nastro.
   b. Se $\delta(r, a) = (s, b, L)$, scrive la codifica binaria di b sul nastro e sposta la testina a sinistra di 2k celle.
4) Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora accetta; se la simulazione raggiunge lo stato di rifiuto di M allora rifiuta; altrimenti prosegue con la simulazione dal punto 2.

3. (8 punti) Una *Turing machine con alfabeto ternario* è una macchina di Turing deterministica a singolo nastro dove l'alfabeto di input è $\Sigma = \{0, 1, 2\}$ e l'alfabeto del nastro è $\Gamma = \{0, 1, 2, \sqcup\}$. Questo significa che la macchina può scrivere sul nastro solo i simboli $0, 1$ e blank: non può usare altri simboli né marcare i simboli sul nastro.

Dimostra che ogni linguaggio Turing-riconoscibile sull'alfabeto $\{0, 1, 2\}$ può essere riconosciuto da una Turing machine con alfabeto ternario.

Per risolvere l'esercizio dobbiamo dimostrare che

a) ogni linguaggio riconosciuto da una Turing machine con alfabeto ternario è Turing-riconoscibile

b) ogni linguaggio Turing-riconoscibile sull'alfabeto {0, 1, 2} è riconosciuto da una Turing machine con alfabeto binario

Quindi:

c) Questo caso è semplice: una Turing machine con alfabeto ternario è un caso speciale di Turing machine deterministica a nastro singolo. Quindi ogni linguaggio riconosciuto da una Turing machine con alfabeto binario è anche Turing-riconoscibile.

d) Per dimostrare questo caso, consideriamo un linguaggio L Turing-riconoscibile, e sia M una Turing machine deterministica a nastro singolo che lo riconosce. Questa TM potrebbe avere un alfabeto del nastro Γ che contiene altri simboli oltre a 0, 1, 2 e blank. Per esempio, potrebbe contenere simboli marcati o separatori. Per costruire una TM con alfabeto ternario B che simula il comportamento di M dobbiamo come prima cosa stabilire una codifica ternario dei simboli nell'alfabeto del nastro Γ di M. Questa codifica è una funzione C che assegna ad ogni simbolo a ∈ Γ una sequenza di k cifre binarie, dove k è un valore scelto in modo tale che ad ogni simbolo corrisponda una codifica diversa. Per esempio, se Γ contiene 9 simboli, allora k = 2, perché con 3 bit si rappresentano 9 valori diversi.

La TM con alfabeto ternario B che simula M è definita in questo modo:

B = Su input w:

- Sostituisce $w = w_1w_2 \ldots w_n$ con la codifica ternaria $C(w_1)C(w_2)\ldots C(w_n)$, e riporta la testina sul primo simbolo di $C(w_1)$.
- Scorre il nastro verso destra per leggere k cifre binarie: in questo modo la macchina stabilisce qual è il simbolo a presente sul nastro di M. Va a sinistra di k celle.
- Aggiorna il nastro in accordo con la funzione di transizione di M:
    a. Se $\delta(r, a) = (s, b, R)$, scrive la codifica ternaria di b sul nastro.
    b. Se $\delta(r, a) = (s, b, L)$, scrive la codifica ternaria di b sul nastro e sposta la testina a sinistra di 3k celle.
- Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora accetta; se la simulazione raggiunge lo stato di rifiuto di M allora rifiuta; altrimenti prosegue con la simulazione dal punto 2.

Dimostra che il seguente linguaggio è indecidibile:

$$A_{1010} = \{\langle M \rangle \mid M \text{ è una TM tale che } 1010 \in L(M)\}.$$

Mostriamo che A_{TM} è riducibile ad A_{1010}. Quindi la funzione di riduzione si calcola su una macchina di Turing F che su input <M,w>:

- se x sua stringa è diversa da 1010, rifiuta
- se x sua stringa è uguale a 1010, esegue M su input w
- se M accetta, *accetta*
- se M rifiuta, *rifiuta*
- si restituisce $\langle M_w \rangle$

Essendo funzione di riduzione:

- se <M, w> sta in A_{TM} allora M accetta *w* e anche 1010. Quindi f (<M,w>) $\in A_{1010}$
- se <M, w> non sta in A_{TM} la computazione di M su w non termina o rifiuta, dunque f (<M,w>) $\notin A_{1010}$

Siccome $A_{TM} \leq_m A_{1010}$ allora A_{TM} è indecidibile e similmente A_{1010}.

(a) Mostrare che A è Turing-riconoscibile *se e solo se* $A \leq_m A_{TM}$.
(b) Mostrare che A è decidibile *se e solo se* $A \leq_m 0^*1^*$.

a) Per mostrare che A è Turing-riconoscibile, deve esistere una macchina in grado di riconoscerlo. Partiamo da una TM M che riconosce A per l'appunto; dato che deve esistere una funzione di riduzione, allora esiste una f(w) tale che sia <M,w> $\in A_{TM}$ e accetta tute le stringhe appartenenti ad A, viceversa le rifiuta. In tale modo se A_{TM} è Turing-riconoscibile, può essere mappato tramite una funzione anch'essa Turing-riconoscibile, correttamente.

b) Se A è mappabile da una riduzione, allora necessariamente esisterà una f tale che f(w) sia composta da 01 per accettare e stringhe diverse, ad esempio 01 per rifiutare.

Dato che si ha una riduzione, ci sarà un decisore in grado di far sempre terminare la computazione e dunque una possibile TM D tale che, eseguendo M sull'input w restituisca 01 per aver accettato, altrimenti ritorna 10. A tali condizione, allora f(w)
Viceversa, avendo la riduzione, sappiamo che 0*1* è decidibile e concludiamo che anche A lo sia.

Una *macchina di Turing con reset a sinistra* è una variante delle comuni macchine di Turing, dove la funzione di transizione ha la forma:

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{R, RESET\}.$$

Se $\delta(q, a) = (r, b, RESET)$, quando la macchina si trova nello stato q e legge a, la testina scrive b sul nastro, salta all'estremità sinistra del nastro ed entra nello stato r. Per sapere su quale cella saltare la macchina usa il simbolo speciale $\triangleright$ per identificare l'estremità di sinistra del nastro. Questo simbolo si può trovare solo in una cella del nastro, e non può essere sovrascritto o cancellato. La computazione di una macchina di Turing con reset a sinistra sulla parola w inizia con $\triangleright w$ sul nastro. Si noti che queste macchine non hanno la solità capacità di muovere la testina di una cella a sinistra.

Mostrare che le macchine di Turing con reset a sinistra riconoscono la classe dei linguaggi Turing-riconoscibili.

(a) Mostriamo come convertire una TM con reset a sinistra M in una TM standard S equivalente. S simula il comportamento di M nel modo seguente. Se la mossa da simulare prevede uno spostamento a destra, allora S esegue direttamente la mossa. Se la mossa prevede un $RESET$, allora S scrive il nuovo simbolo sul nastro, poi scorre il nastro a sinistra finché non trova il simbolo $\triangleright$, e riprende la simulazione dall'inizio del nastro. Per ogni stato q di M, S possiede uno stato q_{RESET} che serve per simulare il reset e riprendere la simulazione dallo stato corretto.

S = "Su input w:
1. scrive il simbolo $\triangleright$ subito prima dell'input, in modo che il nastro contenga $\triangleright w$.
2. Se la mossa da simulare è $\delta(q, a) = (r, b, R)$, allora S la esegue direttamente: scrive b sul nastro, muove la testina a destra e va nello stato r.
3. Se la mossa da simulare è $\delta(q, a) = (r, b, RESET)$, allora S esegue le seguenti operazioni: scrive b sul nastro, poi muove la testina a sinistra e va nello stato r_{RESET}. La macchina rimane nello stato r_{RESET} e continua a muovere la testina a sinistra finché non trova il simbolo $\triangleright$. A quel punto la macchina sposta la testina un'ultima volta a sinistra, poi di una cella a destra per tornare sopra al simbolo di fine nastro. La computazione riprende dallo stato r.
4. Se non sei nello stato di accettazione o di rifiuto, ripeti da 2."

(b) Mostriamo come convertire una TM standard S in una TM con reset a sinistra M equivalente. M simula il comportamento di S nel modo seguente. Se la mossa da simulare prevede uno spostamento a destra, allora M può eseguire direttamente la mossa. Se la mossa da simulare prevede uno spostamento a sinistra, allora M simula la mossa come descritto dall'algoritmo seguente. L'algoritmo usa un nuovo simbolo $\triangleleft$ per identificare la fine della porzione di nastro usata fino a quel momento, e può marcare le celle del nastro ponendo un punto al di sopra di un simbolo.

M = "Su input w:

1. Scrive il simbolo $\triangleleft$ subito dopo l'input, per marcare la fine della porzione di nastro utilizzata. Il nastro contiene $\triangleright w \triangleleft$.
2. Simula il comportamento di S. Se la mossa da simulare è $\delta(q, a) = (r, b, R)$, allora M la esegue direttamente: scrive b sul nastro, muove la testina a destra e va nello stato r. Se muovendosi a destra la macchina si sposta sulla cella che contiene $\triangleleft$, allora questo significa che S ha spostato la testina sulla parte di nastro vuota non usata in precedenza. Quindi M scrive un simbolo blank marcato su questa cella, sposta $\triangleleft$ di una cella a destra, e fa un reset a sinistra. Dopo il reset si muove a destra fino al blank marcato, e prosegue con la simulazione mossa successiva.
3. Se la mossa da simulare è $\delta(q, a) = (r, b, L)$, allora S esegue le seguenti operazioni:
   - 3.1 scrive b sul nastro, marcandolo con un punto, poi fa un reset a sinistra
   - 3.2 Se il simbolo subito dopo $\triangleright$ è già marcato, allora vuol dire che S ha spostato la testina sulla parte vuota di sinistra del nastro. Quindi M scrive un blank e sposta il contenuto del nastro di una cella a destra finché non trova il simbolo di fine nastro $\triangleleft$. Fa un reset a sinistra e prosegue con la simulazione della prossima mossa dal nuovo blank posto subito dopo l'inizio del nastro. Se il simbolo subito dopo $\triangleright$ non è marcato, lo marca, resetta a sinistra e prosegue con i passi successivi.
   - 3.3 Si muove a destra fino al primo simbolo marcato, e poi a destra di nuovo.
   - 3.4 se la cella in cui si trova è marcata, allora è la cella da cui è partita la simulazione. Toglie la marcatura e resetta. Si muove a destra finché non trova una cella marcata. Questa cella è quella immediatamente precedente la cella di partenza, e la simulazione della mossa è terminata
   - 3.5 se la cella in cui si trova non è marcata, la marca, resetta, si muove a destra finché non trova una marcatura, cancella la marcatura e riprende da 3.3.
4. Se non sei nello stato di accettazione o di rifiuto, ripeti da 2."

Give an implementation-level description of a Turing machine that, from an input $\#w_1\#w_2\#$ (where w_1 and w_2 are strings over $\{0, 1\}$, with $|w_1| = |w_2| > 0$), compute **one** of the following functions:

1. The logical AND (x AND y = 1 if and only if $x = y = 1$);
2. The logical OR (x OR y = 0 if and only if $x = y = 0$);
3. The logical XOR (x XOR y = 1 if and only if $x \neq y$).

Those functions are **bit-by-bit** operations. So, for instance, 111 AND 101 = 101; 000 OR 101 = 101; 111 XOR 101 = 010.

Answer:

L'idea è, partendo dalla specifica funzione, crearsi delle TM che riconoscano effettivamente tutto ciò e scrivano in output su un secondo nastro il risultato dell'operazione bit a bit.

Per descrivere la AND, usiamo M_{AND} che sull'input $\#w_1\#w_2\#$

- si va al primo carattere dopo la #. Qui seguono due casi (se 0/1); se si ha un blank, abbiamo raggiunto la fine della computazione ed accetta
- se si vede uno 0, si scrive uno 0 sul secondo nastro e il carattere viene rimpiazzato sul primo da uno #, muovendosi a destra al primo carattere dopo un #, per poi riportarsi alla cella più a sinistra
- se si vede un 1, l'operazione di AND normalmente non comporterebbe scrittura, ma pone un # sul carattere 1 e si muove a destra al primo carattere dopo il #. Se anche qui è uno 0, lo scrive e si sposta sul secondo nastro, altrimenti per un 1 eseguirà la stessa operazione, vedendo che si tratta sempre di un 1 e coerentemente con l'operazione di AND lo scrive sul nastro. Dopodiché si sposta alla cella più a sinistra. Si ripete poi dal primo passo.

Per descrivere la OR, usiamo M_{OR} che sull'input $\#w_1\#w_2\#$

- si va al primo carattere dopo la #. Qui seguono due casi (se 0/1); se si ha un blank, abbiamo raggiunto la fine della computazione ed accetta
- se si vede un 1, si scrive un 1 sul secondo nastro e sul primo viene rimpiazzato il carattere con un #, spostandosi a destra al primo carattere dopo il # e rimpiazzandolo proprio con un #
- se si vede un 0, l'operazione di OR normalmente non comporterebbe modifiche, ma pone un # sul carattere 0 e si muove a destra al primo carattere dopo il #. Se anche qui è uno 0, lo scrive e si sposta a destra sul secondo nastro, altrimenti per un 1 eseguirà la stessa operazione, scrivendo un # nel primo nastro. Si ripete poi dal primo passo.

Per descrivere la XOR, usiamo M_{XOR} che sull'input $\#w_1\#w_2\#$

- si va al primo carattere dopo la #. Qui seguono due casi (se 0/1); se si ha un blank, abbiamo raggiunto la fine della computazione ed accetta
- se si vede un 0, si rimpiazza il carattere con # sul primo nastro e sul secondo, se è uno 0, scrive proprio 0 sul secondo nastro e si sposta a destra su di esso; altrimenti scrive 1 ed esegua la medesima cosa, rimpiazzando il carattere con un # sul primo nastro
- se si vede un 1, rimpiazza il carattere con un # e si sposta a destra al primo carattere dopo il #. Se vede un 1, scrive 0 sul secondo nastro, si muove a destra su di esso; altrimenti vede 1 e scrive 0 a seconda del caso, rimpiazzando il carattere con # sul primo nastro e spostandosi tutto a sinistra

Recall that the complement of a language L over Σ is $\overline{L} = \Sigma^* \setminus L$.

1. Prove that decidable languages are closed under complement.
2. Prove that Turing-recognizable languages are **not** closed under complement.

1) L'operazione di complemento semplicemente viene descritta da una TM $\underline{M}$ che accetta il complemento di M; dunque se M rifiuta, $\underline{M}$ *accetta*, altrimenti M accetta ed $\underline{M}$ rifiuta
2) Ragioniamo per contraddizione. Sia L un linguaggio Turing-riconoscibile, allora anche $\underline{L}$ lo è. Possiamo usare le macchine di Turing che riconoscono L e $\underline{L}$ per decidere L: si eseguono entrambe le TM in parallelo (in pratica, alternandole ad ogni passo di computazione), se quella che riconosce L accetta allora *accetta*, e se quella che riconosce $\underline{L}$ accetta allora *rifiuta*.
Questa macchina di Turing è corretta poiché entrambi i linguaggi sono Turing-riconoscibili, ed è un decisore per L. Quindi, se la classe di lingua riconoscibile di Turing è chiusa sotto complemento, allora ogni linguaggio Turing-riconoscibile è decidibile. Questa è una contraddizione, dal momento che A_{TM} è riconoscibile e indecidibile da Turing.

Prove that the following language is undecidable:

$L = \{< M, q, w > \mid M$ is a Turing machine, q is a state of M, and M reaches q in its computation on $w\}$.

Assumiamo per contraddizione il linguaggio sia decidibile e che sia descritto da una funzione di riduzione f(x che si calcola su una macchina di Turing F su input <M,w> con questi passi:

- si esegue M su w
- se x sua stringa è diversa da w, rifiuta
- se x sua stringa è uguale a w, esegue M e se la trova, *accetta*
- se M rifiuta, *rifiuta*
- si restituisce <M_w>

Essendo funzione di riduzione:

- se <M, w> sta in A_{TM} allora M accetta *w*. Quindi f (<M,w>) $\in A_{TM}$
- se <M, w> non sta in A_{TM} la computazione di M su w non termina o rifiuta, dunque f (<M,w>) $\notin A_{TM}$

Siccome $A_{TM} \leq_m A_L$ allora A_{TM} è indecidibile.

Prove that the following language is undecidable:

$$L = \{< M_1, M_2 > \mid M_1 \text{ and } M_2 \text{ are Turing machines, and } L(M_1) \subseteq L(M_2)\}.$$

Assumiamo per contraddizione il linguaggio sia decidibile e che sia descritto da una funzione di riduzione f(x che si calcola su una macchina di Turing F su input <M,w> con questi passi:

- si esegue M_1 ed M_2 su w
- la stringa di M_1 deve comparire in M_2; dunque la TM deve rifiutare tutti gli input ad eccezione di <M,w>
- se M_1 accetta, *accetta,* altrimenti *rifiuta*
- si restituisce $<M_w>$

Essendo funzione di riduzione:

- se <M, w> sta in A_{TM} allora M accetta *w*. Quindi f (<M,w>) $\in A_{TM}$ e in particolare L(M) $\subseteq \emptyset$
- se <M, w> non sta in A_{TM} la computazione di M su w non termina o rifiuta, dunque f (<M,w>) $\notin$-A_{TM} e L(M) /$\subseteq \emptyset$

Siccome $A_{TM} \leq_m A_L$ allora A_{TM} è indecidibile.

Show that the following predicates are undecidable:

1. $M_1(x, y) =$ "$\text{Dom}(\phi_x) = \text{Dom}(\phi_y)$"
2. $M_2(x) =$ "$\phi_x(x) = 0$"
3. $M_3(x, y) =$ "$\phi_x(y) = 0$"
4. $M_4(x, y) =$ "$x \in \text{Ran}(\phi_y)$"
5. $M_5(x) =$ "ϕ_x is total and constant"
6. $M_6(x) =$ "$\text{Dom}(\phi_x) = \emptyset$"
7. $M_7(x) =$ "$\text{Ran}(\phi_x)$ is infinite"
8. $M_8(x) =$ "$\phi_x = g$" where g is a fixed computable function

Si forniscono soluzioni degli es. che centrano con noi:

1. Let e be an index for the constant zero function. This means that $\phi_e = \mathbb{O}$, and so $\text{Dom}(\phi_e) = \mathbb{N}$. Let $M_{\text{tot}}(x) =$ "$\phi_x$is total". We have:

   $$M_{\text{tot}}(x) \iff \text{Dom}(\phi_x) = \mathbb{N} \iff \text{Dom}(\phi_x) = \text{Dom}(\phi_e) \iff M_1(x, e)$$

   This shows that if we could decide M_1, we could also decide M_{tot}. But we know that the latter is not decidable. Hence, M_1 is not decidable either.

2. To show that M_2 is undecidable, we use the diagonal method. Consider the function:

   $$f(x) = \begin{cases} 1 & \text{if } \phi_x(x) = 0 \\ 0 & \text{otherwise} \end{cases} = \begin{cases} 1 & \text{if } M_2(x) \text{ holds} \\ 0 & \text{otherwise} \end{cases}$$

   Notice that f is the characteristic function of $M_2(x)$. Now, f cannot be equal to any computable function ϕ_x, since by construction $f(x) \neq \phi_x(x)$. So, f is undecidable. Since f is the characteristic function of M_2, this means that M_2 is undecidable.

4. Il Teorema di Rice tratta dell'indecidibilità delle proprietà dei linguaggi RE (cioè riconosciuti dalle Macchine di Turing). Considerate la seguente proprietà P_{REG}: il linguaggio è Regolare. Date la definizione del corrispondente linguaggio $L_{P_{REG}}$. Successivamente, spiegate se questo linguaggio è indecidibile o no. Probabilmente vi servirà il Teorema di Rice.

Assumiamo per contraddizione che P sia un linguaggio decidibile che soddisfa le proprietà e che R_P sia una TM che decide P. Mostriamo come decidere A_{TM} usando R_P costruendo una TM S.
Per prima cosa, si deve dire che P è un linguaggio non triviale; dunque, deve avere una possibile descrizione del linguaggio. Essendo regolare possiamo simularlo con un DFA tale che accetti avanzando regolarmente tutte le sue stringhe. P poi deve essere proprietà del linguaggio, dunque M_2 deve decidere le stringhe di M_1.
Essendo non triviale, si crea S che su input w:

- Usa m per costruire M_w
- M_w avanza su w. Se si ferma, significa che il linguaggio non è regolare e dunque trivialmente segnala NO come output.
- Altrimenti sfrutta R_P che in maniera non triviale cerca di decidere il linguaggio. Se accetta, regolarmente, accetta.

M_w accetta se e solo se P accetta, come proprietà del suo linguaggio.

3. Since $M_2(x) = M_3(x,x)$, if M_3 was decidable so would be M_2. But we have just seen that M_2 is undecidable.

6. **Strategy 1.** Let $f_\emptyset$ denote the function which is always undefined. Notice that $f_\emptyset$ is the only function with an empty domain, that is, $\mathrm{Dom}(\phi_x) = \emptyset \iff \phi_x = f_\emptyset$. So, if we let $\mathcal{B} = \{f_\emptyset\}$, then we have: $M_6(x) \iff \mathrm{Dom}(\phi_x) = \emptyset \iff \phi_x = f \iff \phi_x \in \mathcal{B}$. Since $\mathcal{B}$ is a non-empty and non-total class of computable functions, Rice's theorem applies. So, M_6 is undecidable.

   Strategy 2. We can get to the same result by using the reduction method. We do this by reducing the undecidable predicate $M_{\overline{K}}(x) =$ "$\phi_x(x)\uparrow$" to M_6. For this, we define a function $f(x,y)$ as follows:

$$f(x,y) = \begin{cases} 0 & \text{if } \phi_x(x)\downarrow \\ \text{undefined} & \text{if } \phi_x(x)\uparrow \end{cases}$$

   This function is computable by Church's thesis (a procedure to compute f should be given for this, but this procedure goes in the usual manner). By the s-m-n theorem, we have a total computable k such that $\phi_{k(x)}(y) = f(x,y)$.
   We claim that $M_{\overline{K}} \leq^k M_6$, that is, that $\forall x \in \mathbb{N}$, $M_{\overline{K}}(x) \iff M_6(k(x))$. If this claim is true, since $M_{\overline{K}}$ is undecidable, it follows that M_6 is undecidable as well. So, we are left with the task of showing the claim.

   - Suppose $M_{\overline{K}}(x)$ holds. This means that $\phi_x(x)\uparrow$. So, for all y, $\phi_{k(x)}(y)\uparrow$. Hence, $\mathrm{Dom}(\phi_{k(x)}) = \emptyset$, which means that $M_6(k(x))$ holds.
   - Suppose $M_{\overline{K}}(x)$ doesn't hold. This means that $\phi_x(x)\downarrow$. So, for all y, we have $\phi_{k(x)}(y) = 0$. Hence, $\mathrm{Dom}(\phi_{k(x)}) = \mathbb{N}$, which means that $M_6(k(x))$ does not hold.

Sia

$$L_{XTM} = \{\langle T \rangle : \text{T NON ACCETTA } \langle T \rangle\}$$

Si discuta la decidibilità di tale linguaggio

Sc

Definiamo una funzione di riduzione f(x) tale che:
- q_0 se (T) ∈ L_{XTM}
- q_r se (T) ∉ L_{XTM}

Dunque, osserviamo che se avremo:
- se $f(x)_{X(TM)} = q_0$ allora, dato che deve rifiutare, (T) ∉ L_{XTM}
- se $f(x)_{X(TM)} = q_r$ allora, dato che deve accettare, (T) ∈ L_{XTM}

La definizione di L_{XTM} avrà:
- T, descritta da un solo stato q_0; non essendoci la riduzione, saremo in un assurdo, in quanto l'altra macchina accetterà e contemporaneamente non accetterà l'input, mentre T lo accetta sempre.
  In altre parole, riconoscono una il contrario dell'altra, ricadendo in un assurdo. Più specificamente:

$$\langle T\rangle \in L_{XTM} \iff O_{T_{XTM}}(\langle T\rangle) = q_r \iff \langle T\rangle \notin L_{XTM}$$

- $O_{T_{XTM}}(\langle T\rangle) = q_r \iff \langle T\rangle \in L_{XTM}$ segue dalla definizione di L_{XTM}.
- $O_{T_{XTM}}(\langle T\rangle) = q_r \iff \langle T\rangle \notin L_{XTM}$ segue dall' ipotesi.

Abbiamo che:

$$\langle T\rangle \in L_{XTM} \iff \langle T\rangle \notin L_{XTM}$$

Assurdo! Quindi L_{XTM} non è decidibile.

Esercizio 2 (The Accepting Problem):
Un quesito che ci si pone in modo naturale è il seguente:

> *"Esiste una macchina di Turing in grado di predire se tutte le altre macchine di Turing terminano nello stato di accettazione?".*

Informalmente, ci stiamo chiedendo se esiste una macchina di Turing T che data una qualsiasi altra macchina di Turing M e una parola x, essa riesce a "capire" se la computazione $M(x)$ termina in $\boldsymbol{q_a}$.
Precisiamo che questo quesito è diverso da quello che ci si pone nell' Halting Problem in quanto, in quest'ultimo, ci si chiede:

> *"Esiste una macchina di Turing in grado di predire se le altre macchine di Turing, data una qualsiasi parola in input, terminano ? ".*

La differenza sostanziale tra questi due quesiti è che nel primo siamo interessati a capire, dato un input x, in che **stato finale** terminano le macchine di Turing e nel secondo, invece, si è interessati a capire se, dato un input x, le macchine di Turing **terminano**.

Dopo questa breve precisazione, torniamo all'*Accepting Problem* e verifichiamo se effettivamente esiste una macchina di Turing che permetta di predire l'output di tutte le altre macchine di Turing.

Per rispondere a questo quesito possiamo definire il seguente linguaggio e studiarne l'accettabilità e la decidibilità.
Sia

$$L_{ATM} = \{(\langle T\rangle, x) \ : \ \text{T ACCETTA X}\}$$

Si discuta l'accettabilità e la decidibilità di tale linguaggio.

In questo esercizio è possibile dimostrare che il linguaggio non è decidibile, dunque, detto ciò, crearsi una macchina di Turing decisore e così dimostrare per assurdo che il linguaggio è indecidibile.
In particolare, si discute l'idea del partire da un qualsiasi input e terminare sempre in un particolare stato q_0. L'idea di questo esercizio e che riporto, è la diagonalizzazione.
Immaginiamo infatti di avere una funzione riportata su una matrice; dunque, avremo bisogno di due indici "i" e "j":

$$a_{i,j} = \begin{cases} 1 & \text{SE } T_i(\langle T_j\rangle) \text{ ACCETTA} \\ 0 & \text{SE } T_i(\langle T_j\rangle) \text{ RIGETTA} \end{cases}$$

arrivando poi ad una matrice del tipo:

	$\langle T_1\rangle$	$\langle T_2\rangle$	$\langle T_3\rangle$	$\langle T_4\rangle$	$\langle T_5\rangle$	$\langle T_6\rangle$	...
T_1	0	1	0	0	1	1	...
T_2	1	1	0	1	0	0	...
T_3	0	0	0	0	0	0	...
T_4	1	1	1	1	1	1	...
T_5	0	1	0	0	1	1	...
T_6	1	0	0	1	1	1	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Prendendo la diagonale, avremo una sequenza di 0/1 invertiti casualmente (dato che la macchina dovrà accettare qualsiasi input) e si arriverà ad almeno una sequenza caratteristica che differisce da qualsiasi input della macchina e dalle righe stesse della matrice.
A queste condizioni, L_{XTM} non è decidibile.
Dimostriamo per assurdo quanto detto, creandoci come accennato una macchina che decide T_{XTM}.
Essa sarà descritta da una classica computazione su <M,w>. Essendo letteralmente l'opposto dell'halting problem, allora la simulazione su T_{ATM} con input <<P>, <P>>:

- se T_{ATM} accetta, allora *rifiuta*
- se T_{ATM} rifiuta, allora *accetta*

La macchina così ottenuta sarebbe un decisore, ma è di fatto un assurdo con quanto dimostrato prima.
Il linguaggio, dunque, non è decidibile.

Esercizio 3:
Si consideri il seguente linguaggio (versione modificata dell'Halting Problem):

$$L_{BH} = \{(\langle T\rangle, x) : T(x) \text{ TERMINA IN } |x| \text{ PASSI}\}$$

Si dimostri se L_{BH} è decidibile o non decidibile.

Proviamo a definire una TM T che descrive tale linguaggio; la chiave che noi sappiamo è che termina in |x| passi, informazione data come input; abbiamo bisogno di un modo per sapere quali sono questi input.
Ciò è dato dalla TM multinastro che, in uno dei suoi nastri appunto, conserva tutti gli input presenti.
Sul primo nastro si ha l'input, sul secondo avanza la computazione tramite simulazione di T sulla stringa x mentre sul terzo nastro potremo leggere e/o scrivere.
A seconda degli input che io leggo, l'unico modo che ho di capire se ho letto un particolare simbolo di input è di usare una sorta di mark, per esempio •. Ogni volta mi sposto dopo il simbolo appena messo e capisco se l'ho letto e quante volte l'ho letto. Prima o poi arriverò ad un blank; se un numero |x| di simboli sono stati posti, la macchina accetta, altrimenti rifiuta.
Di fatto così funziona:

$$O_{T_{BH}}(\langle T\rangle, x) = \begin{cases} q_a & \text{SE } (\langle T\rangle, x) \in L_{BH} \\ q_r & \text{SE } (\langle T\rangle, x) \in L_{BH}^c \end{cases}$$

Quindi L_{BH} è un linguaggio decidibile.

Se avessi voluto usare l'idea della riduzione, comunque, di fatto, si sarebbe arrivati ad una macchina che accetta esattamente lo stesso input e il problema sarebbe decidibile.
Essendo |x| una costante, non ho modo di crearmi una riduzione che risolva il problema opposto; avrei semplicemente un sottoinsieme (che significa semplicemente che se io dicessi di avere B che riduce A, B potrà al massimo fare quanto descritto per la TM precedente in |x| + k passi, rispetto ad |x| passi. Essendo entrambi decidibili, il linguaggio rimane decidibile).

Esercizio 4:
Sia $L_1 \subseteq \Sigma^*$ un linguagggio decidibile e sia $L_2 \subseteq \Sigma^*$ un linguaggio accettabile ma non decidibile. Detta T_1 la macchina di Turing che decide L_1 e detta T_2 la macchina di Turing che accetta L_2, si consideri il linguaggio $L \subseteq \Sigma^* \times \mathbb{N}$ di seguito definito:

$$L = \{(x,k) : x \in \Sigma^* \wedge k \in \mathbb{N} \wedge [x \notin L_1 \vee (x \notin L_2 \wedge T_2(x) \text{ RIGETTA IN K PASSI })]\}$$

Si dimostri se L è un linguaggio accettabile o decidibile.

Osserviamo che il linguaggio è decidibile.
ASSUNZIONE: Ogni coppia (x,k) sarà ben formata, ovvero $\forall(x,k)$ avremo sempre che $x \in \Sigma^* \wedge k \in \mathbb{N}$. Quest'assunzione serve per facilitare l'analisi del linguaggio e può essere, chiaramente, rilassata.

Per argomentare tale claim basta osservare che possiamo definire il linguaggio L come l'unione di due linguaggi:

$$L_a = \{(x,k) : x \in \Sigma^* \wedge k \in \mathbb{N} \wedge \ x \in L_1^c\}$$
$$L_b = \{(x,k) : x \in \Sigma^* \wedge k \in \mathbb{N} \wedge (x \in L_2^c \wedge T_2(x) \text{ RIGETTA IN K PASSI}\}$$

Osserviamo che $L = L_a \cup L_b$.
Banalmente L_a è un linguaggio decidibile in quanto può essere deciso definendo una macchina di Turing T_a che esegue le seguenti operazioni:

1) Simula la macchina di Turing T_1 con input x se:
   - $T_1(x)$ Accetta allora T_a **Rigetta**
   - $T_1(x)$ Rigetta allora T_a **Accetta**

Osserviamo esplicitamente che la simulazione a il punto 1) termina sempre in quanto L_1 è un linguaggio decidibile.
Discutiamo ora L_b. Esso è un linguaggio decidibile, argomentiamo:
Mostriamo che esiste una macchina di Turing T_b a 3 nastri che decide L_b. Sia T_b definita come segue:

N_1) Input (x,k)

N_2) Simulazione della computazione $T_2(x)$

N_3) codifica unaria del numero k

Descriviamo, ora, il funzionamento della macchina T_b

- Su Input (x,k)

1) Scrivi k in unario sul nastro N_3.
2) Posiziona la testina del nastro N_3 sul primo $\square$ a sinistra prima del primo 1.
3) Simula $T_2(x)$ e ad ogni passo della simulazione sposta la testina di N_3 di una posizione a destra.

- Se

  a) $T_2(x)$ Accetta e sul nastro N_3 legge 1 allora T_b **Rigetta**

  b) $T_2(x)$ Rigetta e sul nastro N_3 legge un 1 allora T_b **Accetta**

  c) $T_2(x)$ Non è terminata e sul nastro N_3 legge $\square$ allora T_b **Rigetta**

Osserviamo esplicitamente che tale macchina T_b riesce a decidere L_b. Ricordiamo che L_b è il linguaggio composto delle parole che appartengono a L_2^c tali che la computazione di $T_2(x)$ rigetta in k passi. Sappiamo che L_2 è accettabile ma non decidibile, il che significa che esiste una macchina di Turing in grado di accettare $\forall x \in L_2$ e che però per quanto riguarda le parole $x \in L_2^c$ per la computazione $T_2(x)$ non sappiamo cosa accade, la macchina T_2 potrebbe rigettare o non terminare. Quindi se $x \in L^c$ e $T_2(x)$ rigetta (per qualche mistico motivo) entro k passi allora T_b Accetta in quanto $x \in L_b$. Se invece $x \in L_2^c$ ma $T_2(x)$ non rigetta entro k passi allora T_b rigetta in quanto $x \in L_b^c$ (gli altri casi si possono dedurre dalla macchina di Turing descritta sopra).

Dopo questa breve analisi, possiamo studiare il linguaggio L che abbiamo definito come $L = L_a \cup L_b$, abbiamo mostrato che sia L_a *che* L_b sono linguaggi decidibili, quindi possiamo dire che L è un linguaggio decidibile. Definiamo la macchina di Turing che decide L:

- Input (x,k)

1) simula $T_a(x, k)$ se T_a accetta allora **Accetta**, se rigetta esegui passo 2)

2) Simula $T_b(x, k)$ se T_b accetta allora **Accetta**, se T_b rigetta allora **Rigetta**

1. AcceptIllini := $\{\langle M \rangle \mid M$ accepts the string `ILLINI`$\}$
2. AcceptThree := $\{\langle M \rangle \mid M$ accepts exactly three strings$\}$
3. AcceptPalindrome := $\{\langle M \rangle \mid M$ accepts at least one palindrome$\}$

1) Supponiamo che esista un algoritmo che eseguendo una simulazione sull'input <M, w>:
- esegue M sul nastro
- se questa macchina trovasse la stringa ILLINI allora M' come codifica, accetta, assumendo che si ferma sull'input w
- se supponiamo che la macchina non si fermi, allora M' non accetta la stringa ILLINI e il particolare ogni stringa sarà diversa, rigettando l'input su M'

L'assurdo ricade in questo; ci saranno due casi opposti per il cui decisore sarebbe corretto (seguendo la logica dell'halting problem; se si ferma accetta il linguaggio giusto; se non si ferma rifiuta tutte le altre stringhe e comunque sarebbe decisore). Quindi in entrambi i casi il linguaggio è indecidibile.

Detto in tre parole per gli altri due:

2) Supponiamo che esista un algoritmo che eseguendo una simulazione sull'input <M, w>:
- esegue M sul nastro e sull'input w

- se questa macchina trovasse esattamente 3 stringhe, si ferma ed accetta la propria codifica, fermandosi
- se questa macchina non si ferma, troverà meno di 3 stringhe o più di 3 a seconda del caso, rifiutando.

In entrambi i casi si avrebbe un decisore, che è assurdo.

3) Supponiamo che esista un algoritmo che eseguendo una simulazione sull'input <M, w>:
- esegue M sul nastro e sull'input w
- se questa macchina trovasse esattamente il palindromo della stringa, si ferma ed accetta la propria codifica, fermandosi
- se questa macchina non si ferma, troverà almeno la stringa stessa e continuerà a computare non fermandosi o rifiutando

In entrambi i casi si avrebbe un decisore, che è assurdo.

3.15 Show that the collection of decidable languages is closed under the operation of

- [A]**a.** union.
- **b.** concatenation.
- **c.** star.
- **d.** complementation.
- **e.** intersection.

a) Mostriamo l'idea per l'unione. L'idea è di considerare tutte le stringhe per esempio di due linguaggi e fare in modo siano riconosciuti da due macchine diverse. Nel caso dell'unione prendiamo su due linguaggi A e B, due TM M_A ed M_B. L'idea è di avere un input da entrambe le parti un insieme di input, con una computazione che avanza in parellelo.
   - Le due macchine M_A ed M_B agiscono individualmente
   - Se almeno una delle due accetta, l'altra lo farà entro un numero finito di passi, in quanto comprendono lo stesso linguaggio, altrimenti se una rifiuta l'altra o rifiuta o andrà in loop

   Quindi la classi dei linguaggi decidibili è chiusa per unione.

b) L'idea della concatenazione è avere sempre due macchine M_1 ed M_2 tali che avremo una TM M' che sarà definita come concatenazione.
   M' su input w:
   - divide *w* in una serie di parti affinché
   - la prima parte sarà eseguita da parte della prima macchina; se accetta, *accetta sulla prima macchina*, altrimenti *rifiuta*
   - la prima parte sarà eseguita da parte della prima macchina; se accetta, *accetta sulla seconda macchina*, altrimenti *rifiuta*
   - l'accettazione totale si ha quando entrambe le macchine accettano

   Quindi la classi dei linguaggi decidibili è chiusa per concatenazione.

c) L'idea dello star è di considerare tutte le possibili stringhe e accettarle se tutte accettano.
   M' su input w:
   - divide *w* in una serie di parti affinché
   - se M accetta ciascuna stringa e, avendole marcate una per una, arriva a fine nastro, accetta, altrimenti rifiuta

   Quindi la classi dei linguaggi decidibili è chiusa per l'operazione star.

d) L'idea del complemento è letteralmente di accettare la stringa opposta e sappiamo essere vero

e) L'idea dell'intersezione di avere due macchine, ciascuna sul suo linguaggio, in maniera tale da avere:
   M' = Su input w:
   - Eseguo M_1 su w; se M_1 rifiuta, allora rifiuta
   - Eseguo M_2 su w; se M_2 rifiuta, allora rifiuta

   Se entrambi sono accettati, accetta, altrimenti rifiuta.
   Quindi la classi dei linguaggi decidibili è chiusa per intersezione.

3.16 Show that the collection of Turing-recognizable languages is closed under the operation of

[A]**a.** union.
b. concatenation.
c. star.
d. intersection.
e. homomorphism.

Mostrati i decidibili, si descrivono rapidamente tutte le operazioni:
- unione, si hanno due macchine M_A ed M_B tali che le eseguiamo alternatamente passo per passo. Se entrambe accettano, si accetta, altrimenti basta averne una che rifiuta e l'altra rifiuta o va in loop
- concatenazione, si hanno due macchine M_A ed M_B tali da operare su una stringa divisa in parti in maniera non deterministica, scorrendo il nastro. Se eseguendo s_1 e s_2 su M_A ed M_B una delle due si ferma, rifiuta, altrimenti accetta se entrambe accettano
- star, si hanno due macchine M_A ed M_B tali da operare su una stringa divisa in parti in maniera non deterministica, scorrendo il nastro. Se eseguendo $s = s_1, s_2, \ldots s_n$ allora se accettate tutte, la macchina accetta, altrimenti no
- intersezione, si hanno due macchine M_A ed M_B tali da operare su una stringa divisa in parti in maniera non deterministica, scorrendo il nastro. Se eseguendo s_1 e s_2 su M_A ed M_B una delle due si ferma, rifiuta, altrimenti accetta se entrambe accettano. Se una delle due si interrompe nel mentre, si rifiuta tutta la stringa, essendo intersezione
- omomorfismo, quindi tutte le stringhe devono corrispondere all'esecuzione da parte della TM. Se ciò avviene, allora si accetta, altrimenti rifiuta. Si andrà quindi ad eseguire un merge di tutte le esecuzioni sull'input da parte della macchina stessa e la TM capisce se è andata a buon fine o meno.

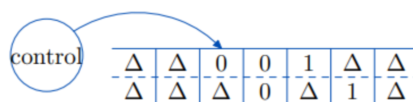
A Jittery TM is one that always writes a different symbol to the one it has just read. Show that a Jittery TM can simulate a standard TM.

For each symbol in Γ, add a copy. Then for each move of the standard TM, the Jittery TM makes two moves: it first writes the duplicate symbol, staying put but going to a temporary state; then it writes the real symbol and moves to the correct state.

For example, the transition $\delta(q, 0) = (r, 0, L)$ becomes $\delta(q, 0) = (q', 0', S)$ and $\delta(q', 0') = (r, 0, L)$.

A ***2-track TM*** is one where there are two symbols in each cell, an upper one and a lower one.

One way to simulate this, is to create a new alphabet: each letter of the alphabet represents a pair of symbols.



Esercizi Linguaggi riducibili tramite funzione ed indecidibili/Riducibilità

1 Usa una riduzione mediante funzione per dimostrare che $ALL_{TM} = \{\langle M \rangle \mid M$ è una TM tale che $L(M) = \Sigma^*\}$ è indecidibile.

2 Sia $X = \{\langle M, w \rangle \mid M$ è una TM a nastro singolo che non modifica la porzione di nastro che contiene l'input $w\}$. Dimostra che X è indecidibile.

3 Se $A \leq_m B$ e B è un linguaggio regolare, allora ciò implica che A è un linguaggio regolare? Perché sì o perché no?

4 Mostra che A_{TM} non è riducibile mediante funzione a E_{TM}. In altre parole, dimostra che non esiste una funzione calcolabile che riduce A_{TM} ad E_{TM}.

5 Mostra che se A è Turing-riconoscibile e $A \leq_m \overline{A}$, allora A è decidibile.

1) Supponiamo che ALL_{TM} sia decidibile e che esista una TM M che decida questo linguaggio. A questo applicheremo una funzione di riduzione. Pertanto, decidiamo di strutturare:
   M = Su input w, dove w è una stringa:
   - se la stringa appartiene al linguaggio (quindi a Σ*) *accetta*
   - altrimenti *rifiuta*, che significa che L(M) = ∅

   Per poter dimostrare che è indecidibile, abbiamo ora costruito un decisore. Dato che deve essere indecidibile, ora mostriamo una macchina A_{TM} che accetta l'opposto; se ciò avviene, come capita, il linguaggio è indecidibile. Quindi per decidere A_{TM}:
   M' = Su input <M, w>, dove w è una stringa ed M è una TM:
   - esegue R su input <M'>
   - se R accetta, allora *accetta*, altrimenti *rifiuta*

   Dato che A_{TM} è indecidibile, R non può esistere e dunque ALL_{TM} è indecidibile.
2) Supponiamo che X sia decidibile e che esista una TM M che decida questo linguaggio. A questo applicheremo una funzione di riduzione. Pertanto, decidiamo di strutturare:
   M = Su input w, dove w è una stringa:
   - comincio a scorrere il nastro
   - se trovo *w*, lo marco e torno all'inizio del nastro
   - una volta esaurito tutto l'input, *accetta* se *w* è stato marcato
   - altrimenti *rifiuta*

   Abbiamo costruito un decisore; ora andiamo ad applicare la riduzione, usando il decisore appena creato per costruire una TM N che decide A_{TM}.
   Questa macchina N, su input <M, w>, dove M è una TM e w è una stringa:
   - simula <M, w> sul nastro
   - marca la fine destra dell'input con un simbolo
   - lo copia sul nastro dopo il blank
   - si simula M su w'
     - se M accetta, si scrive qualsiasi carattere sulla prima parte del nastro ed *accetta*
     - altrimenti *rifiuta*

   Dando in output <M, w>, avremo che l'ultima macchina accetta l'esatto opposto di quanto voluto; abbiamo dimostrato che esiste un decisore tale da accettare non solo la stringa w ma anche tutte le altre stringhe e non modifica mai l'input; dunque A_{TM} è deciso, ma è una contraddizione sulla base di quanto detto. A_{TM} può quindi dirsi indecidibile.

3) Se A $\leq_m$ B e B è un linguaggio regolare, dobbiamo verificare se A possa essere anch'esso un linguaggio regolare. Però semplicemente si potrebbe avere il caso in cui B comprende già tutte le stringhe di A ed A sia non regolare.

A tale scopo facciamo l'esempio di avere A = $\{0^n1^n \mid n >=0\}$ e B = {1}. Dato che sono entrambi decidibili, si ha semplicemente la costruzione per entrambi di una TM che simula sul nastro sulla stringa w ed accetta qualora riconosca esattamente il linguaggio posto da A e da B. Notiamo inoltre che B riconosce tutte le stringhe di A e B è regolare; A invece non lo è.
Si dimostra quindi che B risolve tutti i problemi di A, ma ciò non implica che A sia necessariamente regolare.

4) Essendo A un linguaggio decidibile, anche A_{TM} lo è e similmente E_{TM} è indecidibile. Dobbiamo quindi dimostrare che non esiste una funzione che passando per il test del vuoto decide il linguaggio di A.
Verificando il funzionamento di A, intesa con una TM M:
M = Su input w, dove w è una stringa:
- se w appartiene al linguaggio, quindi a Σ*, accetta
- se w non appartiene al linguaggio, *rifiuta*

Convertiamo quindi ad una funzione di riduzione f che ragiona in questo modo:
M' = Su input <M, w> dove M è una TM e w è una stringa:
- prendendo in input la macchina precedente, dovremmo accettarne almeno una stringa e quando M accetta, *accetta*
- altrimenti *rifiuta*

In output riceveremmo una macchina decisore M_1 che decide se esiste almeno una stringa, rispetto alla macchina precedente. Siccome E_{TM} misura il test del vuoto, ciò significa che il linguaggio deve essere vuoto e deve esistere una funzione di riduzione che accetta quando il linguaggio sia vuoto; noi tuttavia abbiamo dimostrato che il linguaggio non può essere vuoto, ma accetterebbe come visto a lezione $\underline{E_{TM}}$. Siccome la decidibilità non è affetta dal complemento, non esiste la riduzione e il linguaggio è indecidibile.

5) Qui dobbiamo mostrare due cose:
- A è Turing-riconoscibile
- A $\leq_m$ $\underline{A}$

Partiamo col mostrare che A è ridotto da $\underline{A}$; questo perché, se ciò accade, $\underline{A}$ è Turing-riconoscibile, allora anche A lo è. Essendo che sia il complementare che il normale linguaggio sono Turing-riconoscibili, allora A è decidibile.
Verificando il funzionamento di A, intesa con una TM M:
M = Su input w, dove w è una stringa:
- Eseguiamo M su x. Se tale stringa appartiene ad A, *accetta*
- Se tale stringa non appartiene ad A, *rifiuta*

Abbiamo quindi un decisore per A; vogliamo quindi dimostrare che anche $\underline{A}$ ha un decisore e che quindi accetti le stringhe opposte.
Come tale, costruiamo una TM W su input <M, w>:
- esegue <M, w> sul nastro
- se trova un input $\underline{w}$, *accetta*
- altrimenti *rifiuta*

Ora siccome evidentemente:

$$n \in A \iff f(n) \in A'$$

equivalentemente:

$$n \notin A \iff f(n) \notin A'$$

Avendo dimostrato l'esistenza dei decisori per il complemento e anche per le stringhe normali, vediamo che se esiste una stringa appartenente ad A, equivalentemente la stringa non viene accettata da $\underline{A}$ e vale anche il contrario. Pertanto, per ogni n, A è decidibile.

*5.36 Say that a CFG is *minimal* if none of its rules can be removed without changing the language generated. Let $MIN_{CFG} = \{\langle G \rangle | \ G \text{ is a minimal CFG}\}$.

a. Show that MIN_{CFG} is T-recognizable.
b. Show that MIN_{CFG} is undecidable.

a) Si considera una TM M che su un input qualsiasi riconosca una CFG. Per fare ciò passiamo per mezzo della forma normale di Chomsky.
Se è tale, sappiamo che viene generata con una stringa di lunghezza k in (2k – 1) passi.
Avendo infatti una TM M:
M = Su input <G, w>, dove G è una CFG e w è una stringa (non terminale)
- Creare una grammatica equivalente H in una a forma normale di Chomsky da G
- Si considera n; se n=0, allora la derivazione sarà fatta con un solo passo, altrimenti si controllano tutte le derivazioni con 2n – 1 passi
- Si accettano tutte le derivazioni che contengono la stringa w, altrimenti si *rifiuta*

b) In merito invece all'indecidibilità, prima mostriamo un decisore per questa grammatica, mostrando per contraddizione che MIN_{CFG} è decidibile quando sappiamo che non dovrebbe esserlo. Partiamo dalla macchina sopra descritta.
Similmente, costruiamo il complemento della macchina appena descritta, tale che:
Su input <G>:
- Si costruisce la stessa grammatica, aggiungendo al posto di G la grammatica G' e si aggiunge una nuova regola terminale
- Si manda in output <G', A>
Ciò significa che tutte le stringhe del complementare e della grammatica attuale fanno parte del linguaggio; a noi, tuttavia, interessa solo una porzione ridotta, in particolare quella della forma normale di Chomsky.
Pertanto, il linguaggio è indecidibile, sempre per contraddizione

__Tutorato 6__

1) Decidibilità

L = {a, b, abba}
L(A) = Derivato di A, ε, bba

(Esempio dello shuffle)
$L_1 = a_1 \ a_2 \ a_3$
$L_2 = b_1 \ b_2 \ b_3$

$a_1 \ b_1 \ a_2 \ b_2 \ a_3 \ b_3$

a b a (caso XOR)

w TM ←→ Algoritmo (Capire in anticipo cosa un PC può o non può fare)

Tre possibili comportamenti:
1) accetta
2) rifiuta
3) loop

Si fa l'esempio di un grafo della matematica discreta:

Scritto da Gabriel

(partendo da un vertice e mettendo due nodi, 2/5 e 7/11 come archi)

Discutiamo la *decidibilità* e tutte le proprietà, sapendo di avere una risposta:

1) <B, w>
2) <B> ε-NFA
3) <B> RE
4) L(A) = ∅
5) L(A) = L(B)

1) ALL_{DFA} {<A>|A è un DFA tale che L(A) = sigma*}.
   Mostrare che ALL_{DFA} è un linguaggio decidibile.
   $ALL_{DFA} = \{A_1, A_2, A_3, ...\}$
   $A_i = \{Q_i, q_i, \delta_i, F_i, \sum_i\}$
   Goal: TM, A_i

   Opzione 1): A_i, $\sum_i$
   $\sum_i = \{a, b, c, \ldots\}$
   Possiamo prendere una parola da dare in input all'automa:
   abc → A_i (*accetta* oppure *rifiuta* (perché non accetta la parola nel linguaggio))

   Esiste un w_i che non appartiene $L(A_i)$ e quindi A_i appartiene ad ALL_{DFA}, quindi costruendo costruendo un automa che accetta tutto.

   Esercizio 1

1) A, L(A) = ∅ è decidibile
2) ALL_{DFA} = {<A>, L(A) = ∅}
3) I linguaggi regolari sono chiusi rispetto all'operazione di complementare.
   $A_I \rightarrow \underline{A_i}$

   Basta prendere tutti gli stati accettanti e diventano non accettanti, invertendo il tutto. Questo si può fare in un tempo finito.
   Infatti, alla macchina di Turing si danno in pasto tutti gli stati accettanti.
   Partendo da $Q = \{Q_0, Q_1, Q_2\}$
   E si scorre $F = Q_2$ in un tempo finito
   Partendo dal complementare, verifichiamo se il linguaggio accetta o meno il vuoto.

   Informalmente:
   $A_i \rightarrow \underline{A_i} \rightarrow L(\underline{A_i}) = \emptyset$
   $\underline{A_L}$ è un DFA (perché ha tutte le stringhe)

   Esercizio 2
   S_{REX} = {<R, S> | R, S sono espressioni regolari tali che L(R) è sottoinsieme L(S)}
   $L(R) \subseteq L(S)$
   tale che tutte le stringhe di L(R) stiano in L(S).

   Goal: Dimostrare che S_{REX} è decidibile.
   Opzione 1:
   RE → ε-NFA → DFA
   Converto R → DFA *R* → L (DFA R)
   Converto S → DFA *S* → L (DFA S)
   $\exists\, w_i \ \exists\, L(R)$ tale che $w_i \notin L(S)$

   N = {1, 2, 3, 4, 5 …}

Se un numero sta nei pari, non è detto sia finito.

Se $L(R) \subseteq L(S)$ ogni stringa $\in L(R)$, $w_i \in L(S)$.
Avendo due insiemi A e B (tutto regolare cit.)

1) se $A \subseteq B \rightarrow A \cap B = A$ (perché prendiamo solo gli elementi in comune)
2) RE → ε-NFA → DFA (è finito, prima o poi termina)
3) $L(A) = L(B)$ è decidibile
4) $L(A) \cap L(B)$ è decidibile

TM Informale

1) R, S (due ER in input)
   R, S → A, B (posso farlo in un tempo finito)
2) DFA $A \cap B$ (per ogni linguaggio regolare sta un DFA e viceversa) = C
3) $L(A \cap B) = L(A)$

Tutto permette di risolvere il problema in tempo finito

Esercizio 3
$A_{\varepsilon CFG} = \{G \mid G$ è CFG che genera la parola vuota$\}$
Mostrare che CFG è decidibile.

1)
G → CNF
Il problema è decidibile perché applicando le derivazioni di Chomsky, prima o poi finisce e si arriva alle regole terminale.
Se il linguaggio genera la parola vuota, allora avremo almeno una produzione.

2)
$S \rightarrow \varepsilon$
Si converte la CFG in Chomsky e, facendolo riconoscere ad un automa, dopo un certo numero di passi finisce. Se la regola $S \rightarrow \varepsilon$ è presente, accetta, altrimenti rifiuta.

Se la grammatica non è in CNF, va in loop?

Esercizio 4
$E_{TM} = \{M \mid M$ è una TM tale che $L_M = \emptyset\}$
Mostrare che E_{TM}^C è Turing-riconoscibile

TM* una TM che prende in input altre TM
In realtà sono tutte stringhe che accettano altre stringhe,
$E_{TM} = L\ TM_1, TM_1, \ldots$
TM_i, $\sum_i$ non accetta nessuna stringa

Se noi diamo una qualsiasi stringa i-esima, ogni suo risultato va in R (Reject) oppure va in loop.
$E_{TM}^C = \{TM_1', TM_2'$
TM_i', $\sum_i'$ ed esiste almeno una w_i appartenente a $\sum_i'$
$L(TM_i') \neq$ vuoto
Esiste almeno una stringa appartenente all'alfabeto tale che TM_i' accetta w_i'

1) TM
2) TM^C
   La macchina può andare in loop perché è Turing-riconoscibile; non sappiamo se sarà il vuoto o meno, ma potrebbe non esserlo
   Prendiamo quindi la TM, definiamo il complementare e si inizia a provare.

Basta quindi mi esista almeno una procedura.

Se va in loop comunque va bene perché accetta almeno una parola vuota, ma a noi va bene perché è Turing-riconoscibile.

Turing Machine con alfabeto binario
$\sum = \{0, 1\}$
M = {0, 1, |_|}

Dobbiamo quindi avere altri linguaggi Turing-riconoscibili su $\sum = \{0, 1\}$ che possono avere un simbolo di nastro maggiore.
Per esempio M {0, 1, |_|, *, z, …}

Si citano le TM multinastro che per spostare la testina aumenta il numero di simboli.
$\sum = \{a, b\}$ M = {a, b~~, a*, b*}~~
Possiamo quindi prendere una TM ordinaria e trasformarla in una che riconosce più simboli, diciamo.
Quindi una TM che ha meno capacità viene per forza rispettata da quella che ha più capacità in termini computazionali.

Prendiamo ad esempio:
P = {0, 1, |_|, *, @, …}
|P| = K
Serviranno 2^K codifiche binarie.
Quindi decidiamo noi come si dispongono i simboli e cosa accettano di volta in volta.

Se infatti prendiamo una TM con input:
$w_1w_2w_3$
La codifica binaria aggiunge dei simboli e:
$C(w_1)$ $C(w_2)$ $C(w_3)$
Sul nastro quindi, avendo un blocco di simboli sull'input e ammettiamo, su una generica posizione i-esima, di essere nella precedente. Ammettiamo che per riconoscere noi scorriamo; se ciò accade per *k* posizioni, l'elemento che noi conosciamo, possiamo riconoscere la codifica binaria del simbolo che ci interessa, scorrendo *k* volte e abbiamo letto quindi la parola.

Ci spostiamo a sinistra di *k* posizioni, torniamo indietro e rimodifichiamo le parole una alla volta riscorrendo tutto il nastro, delle corrispettive *k* posizioni.
Partendo da una certa, partendo da un punto, saremo di nuovo nella posizione precedente, ma avanti di uno, per l'introduzione di tutti i simboli.
Quindi su un nastro noi ci scorriamo tutte le posizioni e ammettiamo, con l'introduzione del nuovo simbolo, di essere un simbolo più avanti.

Ritornando indietro rispetto alla scrittura, si sarà sull'ultima cella della parola corretta.
Quindi l'idea è che accettando le k computazioni, shiftiamo di un simbolo e scorrendo accettiamo anche il nuovo simbolo.
2*k è quando si muove alla testina, 2^K è il numero di simboli

5.26 Define a ***two-headed finite automaton*** (2DFA) to be a deterministic finite automaton that has two read-only, bidirectional heads that start at the left-hand end of the input tape and can be independently controlled to move in either direction. The tape of a 2DFA is finite and is just large enough to contain the input plus two additional blank tape cells, one on the left-hand end and one on the right-hand end, that serve as delimiters. A 2DFA accepts its input by entering a special accept state. For example, a 2DFA can recognize the language $\{a^n b^n c^n | n \geq 0\}$.

a. Let $A_{2DFA} = \{\langle M, x\rangle | M$ is a 2DFA and M accepts $x\}$. Show that A_{2DFA} is decidable.

b. Let $E_{2DFA} = \{\langle M\rangle | M$ is a 2DFA and $L(M) = \emptyset\}$. Show that E_{2DFA} is not decidable.

a) Immaginando di avere un DFA che agisce in modo finito con due testine, allora prendendo il problema in questione, immaginiamo di avere appunto in input su TM M:
M = Su input <M, x> dove M è una TM e x è una stringa:

1) Essendoci due testine indipendenti, certamente lavoriamo sullo stesso input ma ha senso lavorare con due nastri, di cui uno di input e uno vero e proprio su cui si scorre, quindi binastro.
2) La testina del primo muove sul nastro di input e, così facendo, verifica l'input in sé, tale da capire se possa essere codificato ed interpretato correttamente; se non viene riconosciuto *rifiuta*
3) Sul secondo nastro scorriamo e copiamo tutto l'input del primo, tale che se il primo si blocca similmente succede anche al secondo.
4) Se il primo arriva alla fine della computazione senza errori, similmente, anche il secondo lo sarà. Se ciò avviene, *accetta* la computazione in un numero finito di passi.

b) Ragioniamo per contraddizione e ammettiamo possa esistere un decisore. Come prima, occorre ragionare con due nastri e verificare se in almeno uno di questi esiste una stringa; nel qual caso rifiuta (dato che deve essere il linguaggio vuoto), altrimenti accetta.
Dunque, immaginando di avere
T = Su input <M>

- Esegue M su input w
  - Controlla se esiste almeno una stringa non vuota sul nastro, se ciò avviene *rifiuta*
  - Scorre fino alla fine il nastro di input copiando sull'altro nastro la configurazione; se ciò avviene fino alla fine senza interruzioni, *accetta*.

Tuttavia, il linguaggio di partenza è indecidibile e non può esistere un decisore.

5.27 A ***two-dimensional finite automaton*** (2DIM-DFA) is defined as follows. The input is an $m \times n$ rectangle, for any $m, n \geq 2$. The squares along the boundary of the rectangle contain the symbol # and the internal squares contain symbols over the input alphabet Σ. The transition function $\delta: Q \times (\Sigma \cup \{\#\}) \longrightarrow Q \times \{L, R, U, D\}$ indicates the next state and the new head position (Left, Right, Up, Down). The machine accepts when it enters one of the designated accept states. It rejects if it tries to move off the input rectangle or if it never halts. Two such machines are equivalent if they accept the same rectangles. Consider the problem of determining whether two of these machines are equivalent. Formulate this problem as a language and show that it is undecidable.

Per mostrare che il problema è indecidibile, dimostriamo che esiste un decisore per questo linguaggio. Dunque, devono essere uguali e avere entrambe stringhe non vuote.
Nel caso di uno dei due, prendiamo la configurazione e in base al linguaggio, A si muove considerando tutti e 4 simboli di input, presi dalla stringa *w*. Se arriva effettivamente ad uno degli stati accettanti, normalmente accetterà; usando una funzione di riduzione su <M, w>:

- Esegue la macchina M su w
- Controlla che ogni input sia nelle quattro transizioni accettate
- Si sposta fino alla fine, dato che la stringa deve essere sempre non vuota e non può fermarsi
- Se effettivamente si ferma in uno degli stati d'arresto indicati, allora *accetta*

- Se non arrivasse ad un'accettazione, similmente, la macchina continua a computare, possibilmente andando in loop; se la stringa è vuota non esiste una forma di accettazione da M a w. Ciò significa che il linguaggio è uguale al vuoto e arriva ugualmente all'accettazione

Già da qui si avrebbe una contraddizione; in entrambi i casi, sapendo che il linguaggio è indecidibile, non può esistere un decisore. Dunque, $EQ_{2DIM\text{-}DFA}$ è indecidibile.

Let

$$f(x) = \begin{cases} 3x+1 & \text{for odd } x \\ x/2 & \text{for even } x \end{cases}$$

for any natural number x. If you start with an integer x and iterate f, you obtain a sequence, $x, f(x), f(f(x)), \ldots$. Stop if you ever hit 1. For example, if $x = 17$, you get the sequence 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Extensive computer tests have shown that every starting point between 1 and a large positive integer gives a sequence that ends in 1. But the question of whether all positive starting points end up at 1 is unsolved; it is called the $3x + 1$ problem.

Suppose that A_{TM} were decidable by a TM H. Use H to describe a TM that is guaranteed to state the answer to the $3x + 1$ problem.

Immaginiamo di costruire una TM che descrive questo problema. Dunque, la macchina M su input x:

- Se x = 1, è caso base e accetta
- Se x è pari allora x = 3x + 1
- Se x è dispari, allora x = x/2
- Si continua ripartendo dal controllo iniziale

L'idea è che potenzialmente la macchina si fermerà, trovando la computazione corretta; in uno scenario più realistico, andrà spesso in loop. Infatti, per poter garantire se esista o meno una risposta al problema, si può ipotizzare di utilizzare una riduzione che, in un certo momento, capisca se il problema computazionalmente ha senso e capire se rappresenta una contraddizione.
Infatti, una funzione f su input <M, w>:

- Prende l'input iniziale su M
- Esegue M su w sulle condizioni dette prima
- Se M non accetta, continua ad eseguire sulle condizioni di prima e la macchina andrà sempre in loop

Dando in output M, la macchina andrà in loop e il decisore arriverà, in un tempo non-deterministico, eventualmente ad una soluzione. Quindi, o si trova una stringa che non sta nel linguaggio o non si trova nulla.

Sia C = {<M>| M è una TM a due nastri che scrive un simbolo non vuoto sul suo secondo nastro quando viene eseguito su un input}. Mostra che A_{TM} si riduce a C. Assumi per motivi di contraddizione che TM R decida C. Costruisci un TM S che usa R per decidere A_{TM}.

S = "Su input <M, w>:
1. Utilizzare M e w per costruire il seguente TM T_w a due nastri.

    Tw = "Su qualsiasi input:

    1. Simulare M su w usando il primo nastro.

    2. Se la simulazione mostra che M accetta, scrivi un simbolo non vuoto sul secondo nastro.

2. Eseguire R su < T_w>per determinare se T_w scrive mai un simbolo non vuoto sul suo secondo nastro.
3. Se R accetta, M accetta w, quindi accetta. Altrimenti, rifiuta".

Let $C_{\mathsf{CFG}} = \{\langle G, k\rangle |\ G$ is a CFG and $L(G)$ contains exactly k strings where $k \geq 0$ or $k = \infty\}$. Show that C_{CFG} is decidable.

Immaginiamo di eseguire una TM M sulla CFG indicata, allora:
- Scorre il nastro seguendo le derivazioni della CFG
- Se abbiamo una derivazione con k passi, accetta regolarmente, altrimenti rifiuta

Se l'input è effettivamente ∞, allora:
- Se M scorre su L<G> e trova che k = ∞, allora accetta, altrimenti rifiuta

Per fare ciò, essendo una grammatica context-free, deve essere progettata in modo tale che le derivazioni siano ricorsive e dunque si generi un loop di k volte, corrispondente ad infinito.
Se effettivamente k sarà uguale al numero di volte indicato, allora *accetta*, altrimenti *rifiuta*
Nel caso di un linguaggio decidibile, siamo in presenza di un loop finito e quindi di un linguaggio non regolare, tale che il loop sia visto come un pumping su *k*.

Let $E = \{\langle M\rangle |\ M$ is a DFA that accepts some string with more 1s than 0s$\}$. Show that E is decidable. (Hint: Theorems about CFLs are helpful here.)

Essendoci più 1 che 0, allora in presenza di un linguaggio CF sarà accettato da un DFA in condizioni di una grammatica che accetta un linguaggio. Tale DFA scorre tutti gli stati usando i simboli presenti; immaginandolo attraverso il PL per CFL allora esisterà sempre un pumping tale che le due parti non vuote rimangano tali e le computazioni siano finite.
Dunque, essendo una TM T che esegue su input <M, w> e se accetta per computazioni finite quella particolare stringa, allora *accetta*, altrimenti *rifiuta*.

Say that a variable A in CFL G is ***usable*** if it appears in some derivation of some string $w \in G$. Given a CFG G and a variable A, consider the problem of testing whether A is usable. Formulate this problem as a language and show that it is decidable.

[A]**4.14** Let $\Sigma = \{0,1\}$. Show that the problem of determining whether a CFG generates some string in 1^* is decidable. In other words, show that

$$\{\langle G\rangle |\ G \text{ is a CFG over } \{0,1\} \text{ and } 1^* \cap L(G) \neq \emptyset\}$$

is a decidable language.

4.14 You showed in Problem 2.18 that if C is a context-free language and R is a regular language, then $C \cap R$ is context free. Therefore, $1^* \cap L(G)$ is context free. The following TM decides the language of this problem.

"On input $\langle G\rangle$:
1. Construct CFG H such that $L(H) = 1^* \cap L(G)$.
2. Test whether $L(H) = \emptyset$ using the E_{CFG} decider R from Theorem 4.8.
3. If R accepts, *reject*; if R rejects, *accept*."

Let $A = \{\langle M\rangle |\ M$ is a DFA that doesn't accept any string containing an odd number of 1s$\}$. Show that A is decidable.

The following TM decides A.

"On input $\langle M\rangle$:
1. Construct a DFA O that accepts every string containing an odd number of 1s.
2. Construct a DFA B such that $L(B) = L(M) \cap L(O)$.
3. Test whether $L(B) = \emptyset$ using the E_{DFA} decider T from Theorem 4.4.
4. If T accepts, *accept*; if T rejects, *reject*."

A★**4.25** Let $BAL_{DFA} = \{\langle M\rangle |\ M$ is a DFA that accepts some string containing an equal number of 0s and 1s$\}$. Show that BAL_{DFA} is decidable. (Hint: Theorems about CFLs are helpful here.)

The language of all strings with an equal number of 0s and 1s is a context-free language, generated by the grammar $S \to 1S0S \mid 0S1S \mid \varepsilon$. Let P be the PDA that recognizes this language. Build a TM M for BAL_{DFA}, which operates as follows. On input $\langle B\rangle$, where B is a DFA, use B and P to construct a new PDA R that recognizes the intersection of the languages of B and P. Then test whether R's language is empty. If its language is empty, *reject*; otherwise, *accept*.

5.35 Say that a variable A in CFG G is ***necessary*** if it appears in every derivation of some string $w \in G$. Let $NECESSARY_{CFG} = \{\langle G, A\rangle |\ A$ is a necessary variable in $G\}$.

a. Show that $NECESSARY_{CFG}$ is Turing-recognizable.
b. Show that $NECESSARY_{CFG}$ is undecidable.

a) Dobbiamo partire da una CFG che usa A in ogni derivazione. Quindi semplicemente, su una CFG che comprende la forma G/A, cioè tale che ogni regola contiene A:
M = Su input <G, A> dove G è una CFG ed A è una regola non terminale:

1) Crea una CFG G/A dove sappiamo che appare ad ogni regola
2) Verifica che ogni regola la contenga e la toglie, lasciando l'input w alla fine con tutte le stringhe tranne che
3) Se abbiamo tolto tutte le A e la grammatica non contiene stringhe (dato che devono comprendere per forza A tutte le derivazioni), allora *accetta*, altrimenti rifiuta.

L'idea è quindi che A sia non terminale, in questo modo la macchina continuerà a cercare ed "eventualmente" arriva ad una soluzione. In altri casi, se la grammatica G non comprendesse A perché non necessaria, continuerà ad avere derivazione e togliendo regole, potremmo non terminare mai il controllo.
Pertanto, $NECESSARY_{CFG}$ è Turing-riconoscibile
b) Per dimostrare che $NECESSARY_{CFG}$ è indecidibile, ammettiamo esista una funzione di riduzione *f* che prende in input una TM M sulla grammatica G
Quindi f:

1) Comincia a creare delle regole del tipo:
   S → A
   A → ε
   Poi, incrementalmente, creerà regole del tipo
   A → Aa
   Ad un certo punto, per il principio della leftmost derivation, arriverà ad una derivazione che comprende una A tante quante sono le derivazioni
2) Se ciò avviene la macchina M esegue su questa e se accetta, *accetta*, altrimenti *rifiuta*

È evidente che A è inutile per ogni regola; semplicemente riotterremmo le stesse derivazioni, ma senza la A.
Inoltre, abbiamo trovato un decisore, in contrasto con l'indecidibilità presente.
Pertanto, $NECESSARY_{CFG}$ è indecidibile.

Say that a CFG is *minimal* if none of its rules can be removed without changing the language generated. Let $MIN_{CFG} = \{\langle G\rangle |\ G$ is a minimal CFG$\}$.

a. Show that MIN_{CFG} is T-recognizable.
b. Show that MIN_{CFG} is undecidable.

a) Se ammettiamo tali condizioni, ci aspettiamo di essere in CNF; pertanto, si compirà una derivazione in 2n – 1 passi partendo da n = |w|.
Infatti, immaginando una TM M che su input *G* una grammatica in CNF e sulla stringa *w*:

- Converto la grammatica presente in CNF

- Se n = |w| diventa n = 0 per ogni derivazione singola, allora tutte le regole singole sono necessarie e composte esclusivamente da elementi unitari; altrimenti si controllano le derivazioni con 2n – 1 passi.
- Se si accettano tutte le derivazioni, la grammatica è corretta e si *accetta*, altrimenti si *rifiuta*

Dato che potenzialmente nella ricerca delle derivazioni si potrebbe andare in loop, allora il linguaggio è Turing-riconoscibile.

b) Supponendo che il linguaggio sia indecidibile, immaginiamo di comporre una funzione di riduzione che prende in input la CFG G e la costruisce con una funzione di riduzione:

1) Sull'input della grammatica G, costruiamo una nuova regola terminale per ogni produzione di G'

    2) Eseguiamo la TM M su G e se alle derivazioni di M si ha una w, *accetta*

    3) Se M rifiuta, *rifiuta*

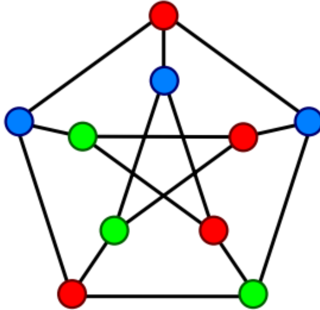
4) Se G aggiunge correttamente ogni regola terminale, allora *accetta* raggiungendo uno stato di accettazione, altrimenti rifiuta

Siccome possiamo accettare qualsiasi stringa che sia presente nel linguaggio, ciò dimostra che la macchina si ferma anche per tutte le regole che possono non essere derivate da G, perché a noi basta aggiungere i simboli terminali; per queste considerazione e dato che $A_{TM} \leq_m MIN_{CFG}$ e A_{TM} è indecidibile, allora MIN_{CFG} è indecidibile.

Esercizi Problemi P/NP

Dal file 19-esercizi

"Colorare" i vertici di un grafo significa assegnare etichette, tradizionalmente chiamate "colori", ai vertici del grafo in modo tale che nessuna coppia di vertici collegati da un arco condivida lo stesso colore. La figura seguente mostra un esempio di colorazione di un grafo con 10 vertici che usa 3 colori (rosso, verde, blu).



Chiamiamo 3-COLOR il problema di trovare, se esiste, una colorazione di un grafo che usa 3 colori diversi. Dimostrate che 3-Color è un problema in NP nel modo seguente:

1) *definite com'è fatto un certificato per 3-COLOR*
2) *definite un verificatore polinomiale per 3-COLOR*

1) L'idea della dimostrazione è di avere un'informazione che capisca se i nodi vicini siano connessi e siano di un colore diverso rispetto al nodo attuale; tanto basterebbe, dato che letteralmente ad ogni nodo sono connessi almeno tre nodi e ciascuno di questi ha un colore diverso rispetto al nodo attuale (tra loro possono anche avere valore uguale).
   Ciò rappresenta il concetto di certificato per questo particolare problema.
2) Consideriamo poi il verificatore V
   V = Su input <G, c>
   1) Controlla se il nodo attuale appartiene al vettore dei nodi, tale che sia connesso ai vicini
   2) Controlla se G contiene tutti i nodi di C
   3) Se il nodo appartiene a G ed è di diverso colore, allora continua ciclicamente a testare
   4) Se tutti accettano, *accetta*, altrimenti *rifiuta*

Considerate il seguente problema, che chiameremo SubsetSum: dato un insieme di numeri interi S ed un valore obiettivo t, stabilire se esiste un sottoinsieme S' ⊆ S tale che la somma dei numeri in S' è uguale a t. Esempio: se S = {4, 11, 16, 21, 27} e t = 25, allora il sottoinsieme S' = {4, 21} `e una soluzione di SubsetSum perché 4 + 21 = 25.
Dimostrate che SubsetSum è un problema in NP nel modo seguente:

1) *definite com'è fatto un certificato per SubsetSum*
2) *definite un verificatore polinomiale per SubsetSum*

1) Il certificato è il sottoinsieme stesso, dato che abbiamo un insieme di numeri e dobbiamo poi capire se appartengono all'insieme dei numeri completo.
2) L'idea di verificatore in tempo polinomiale è la seguente per V:
   V = Su input <<S, t>, c>:
   a. Controlla se c è una collezione di numeri la cui somma è t
   b. Controlla se la somma S contenga tutti i numeri
   c. Se entrambi passano, *accetta*, altrimenti *rifiuta*

Let

$$MODEXP = \{\langle a,b,c,p\rangle \mid a,b,c, \text{ and } p \text{ are positive binary integers such that } a^b \equiv c \pmod p\}.$$

Show that $MODEXP \in$ P. (Note that the most obvious algorithm doesn't run in polynomial time. Hint: Try it first where b is a power of 2.)

Un algoritmo in tempo polinomiale A che risolve il problema è così composto:
M = "Su input <a, b, c, p> dove a, b, c, p sono 4 interi binari:

1) Si calcola x = a mod p inizializzando y ad 1 ed i a 0.
2) Per $b = b_n b_{n-1} \dots b_1 b_0$ si esegue per n + 1 volte
3) Se $b_i = 1$, allora y = y * x mod p; $x = x^2$ mod p; i = i + 1
4) Se y ≡ c(mod p), *accetta*, altrimenti *rifiuta*

L'esecuzione dell'algoritmo ha tempo O(n), pertanto esegue linearmente.

A ***permutation*** on the set $\{1,\dots,k\}$ is a one-to-one, onto function on this set. When p is a permutation, p^t means the composition of p with itself t times. Let

$$PERM\text{-}POWER = \{\langle p,q,t\rangle \mid p = q^t \text{ where } p \text{ and } q \text{ are permutations on } \{1,\dots,k\} \text{ and } t \text{ is a binary integer}\}.$$

Show that $PERM\text{-}POWER \in$ P. (Note that the most obvious algorithm doesn't run within polynomial time. Hint: First try it where t is a power of 2.)

Siccome dobbiamo realizzare delle permutazioni tali che p e q siano delle permutazioni fino a *k* per un certo numero di iterazioni *t*, allora sappiamo che eseguiremo rappresentazione binarie.
Pertanto, q^t può essere scritto come:
$q^t = q^{x_0\ 2\text{^}0} \dots q^{x_n\ 2\text{^}n}$
Da questo si calcola q^{2*j} dove j = 1, 2, … log(t), essendo una cosa esponenziale.
Pertanto, la sostituzione dei valori impiega un'iterazione almeno logaritmica, moltiplicata per un fattore di scala, quindi almeno O(k*log(t)). In conclusione, anche qui, siamo in un problema polinomiale

7.9 A ***triangle*** in an undirected graph is a 3-clique. Show that $TRIANGLE \in$ P, where $TRIANGLE = \{\langle G\rangle \mid G \text{ contains a triangle}\}$.

Una clique/cricca indica che per ogni coppia di vertici, esiste un arco che li collega.
L'dea è che per mostrare che il grafo sia di questo tipo, certamente abbiamo bisogno di triple di vertici riconosciute da una TM, la quale si adopera capendo per ogni arco, se ne esistono almeno 3 a lui connessi.
Quindi, abbiamo:
A = Su input G<V, E>, indicando V un set di vertici e con E una serie di archi:

1) Per ogni tripla di vertici, si controlla se esistono tre archi a lui accoppiati
2) Se per una esiste, si continua per ciascuna di queste.

Il tempo di esecuzione è almeno E per il numero di archi e v^3 avendo almeno una quantità cubica di vertici da controllare; pertanto, il tempo totale è $O(|V|^3|E|)$.

Dal file 20-esercizi NP-Completi:

Esercizio 1

Un circuito Hamiltoniano in un grafo G è un ciclo che attraversa ogni vertice di G esattamente una volta. Stabilire se un grafo contiene un circuito Hamiltoniano è un problema NP-completo.

Un *circuito quasi Hamiltoniano* in un grafo G è un ciclo che attraversa esattamente una volta tutti i vertici del grafo *tranne uno*. Il *problema del circuito quasi Hamiltoniano* è il problema di stabilire se un grafo contiene un circuito quasi Hamiltoniano.

- Dimostra che il problema del circuito quasi Hamiltoniano è in NP
- Dimostra che il problema del circuito quasi Hamiltoniano è NP-hard, usando il problema del circuito Hamiltoniano come problema di riferimento.

1) Abbiamo dimostrato in precedenza che il circuito Hamiltoniano è in NP. Dato che in uno quasi-Hamiltoniano è la stessa cosa meno un vertice, riscriviamo estensivamente la dimostrazione articolando:
N = Su input <G, s, t> dove G è un grafo quasi-hamiltoniano, "s" è un insieme di nodi e t è il certificato:

- prendiamo una lista di numeri, che rappresenta i nodi del grafo
- si controlla per le ripetizioni nella lista, se ne viene trovata una si rifiuta
- si considera che s parta dal primo numero p_1 e t vada fino a $p_m - 1$, dato che non considero l'ultimo vertice nell'attraversamento degli archi
- per ciascun arco tra 1 ed m – 2 si controlla se $<p_i, p_{i+1}>$ induttivamente sia un arco di G. Se ciò accade, tutti i test sono passati

Il ciclo che si genera non include nessun nuovo vertice, infatti ognuno avrà almeno grado 1.
Non sappiamo per certo in quanto tempo polinomiale accada; alcuni sono confonti lineari e sappiamo viene verificato in un tempo al più polinomiale; la selezione stessa potenzialmente è un problema non deterministico.
2) Per rappresentare la struttura del circuito quasi Hamiltoniano come problema NP-Hard, dobbiamo provare che tutti i linguaggi in NP sono riducibili in tempo polinomiale con il problema del circuito quasi Hamiltoniano. L'idea quindi del circuito hamiltoniano è di costruire un grafo G sulla base degli n vertici, per una certa costante k; ne aggiungerà almeno k – 1, ciascuno connesso a tutti gli altri. Dato che il circuito hamiltoniano comprende tutti gli archi, rimarrà sempre almeno un vertice di grado 1 ad ogni computazione polinomiale, considerabile per il circuito Hamiltoniano completo come ciclo a sé stante, continuando a computare.
In particolare, data questa ultima considerazione, matematicamente descriviamo:
$G' = (\{v_1, \dots, v_n\} \cup \{v_1', \dots, v_n'\}$ e $E \cup \{(v_i, v_i') \mid 1 \leq i \leq n\}$
ogni vertice visiterà tutti gli altri con un ciclo $v_i - v_i' - v_i$ (perché almeno un vertice fa ciclo con sé stesso).
Pertanto, il grafo è quasi-hamiltoniano se e solo se G è Hamiltoniano, togliendo il vertice in più.
Se ciò avviene, da in output SI altrimenti manderà in output NO.

oppure

Per completare la prova, dobbiamo dimostrare che è NP-Hard, usando la riduzione di un altro problema NP-completo, in questo caso, sfruttando la riduzione per mezzo del problema del circuito Hamiltoniano.
Dobbiamo, come per il caso precedente, partire da un grafo che chiamiamo G.
Per ogni insieme di vertici, si aggiunge almeno un vertice per ogni iterazione in G
Voglio dimostrare che G ha un circuito hamiltoniano se e solo se H ha un circuito quasi-hamiltoniano.

- Supponiamo che G abbia un circuito hamiltoniano. Aggiungo un vertice senza collegargli nessun arco. Il sottografo certamente contiene un circuito hamiltoniano e raggiungiamo tutti i vertici meno uno. Questo vertice verrà aggiunto successivamente in un ciclo da H, tale che esso colleghi tutti i

vertici e sia considerabile sottoinsieme del precedente (quindi H contiene un ciclo con tutti i vertici considerando anche il vertice tralasciato da G).
- A queste condizioni, G contiene per forza tutti i vertici ed H contiene a sua volta tutta una serie di vertici più quello che non fa parte del ciclo di G. Le due condizioni, come richiesto, vanno di pari passo, affinché G sia hamiltoniano se e solo se H è quasi hamiltoniano.

Dato inoltre che H è praticamente una copia di G, ma solo dei vertici utili (compreso il vertice mancante per G), si ha la correttezza della prova. Inoltre, affermiamo che dato l'insieme di vertici di partenza, la riduzione è corretta, avendo che l'assegnazione di vertici e archi cresce di un fattore costante, nonostante la loro ricerca sia impiegata in tempo lineare. Quindi, anche la riduzione è corretta, dato che il vertice è isolato. Alla luce di tutti questi fatti, anche il problema del circuito quasi-Hamiltoniano è NP-Completo.

Esercizio 2

Considera i seguenti problemi:

SetPartitioning $=\{\langle S\rangle \mid S$ è un insieme di numeri interi che può essere suddiviso in due sottoinsiemi disgiunti S_1, S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in $S_2\}$

SubsetSum $=\{\langle S, t\rangle \mid S$ è un insieme di numeri interi, ed esiste $S' \subseteq S$ tale che la somma dei numeri in S' è uguale a $t\}$

Sappiamo che SetPartitioning è un problema NP-completo.

- Dimostra che SubsetSum è in NP.
- Dimostra che SubsetSum è NP-Hard usando SetPartitioning come problema di riferimento.

1) La dimostrazione di un verificatore V per SUBSET-SUM è articolata in questo modo:
Il certificato è il sottoinsieme stesso, dato che abbiamo un insieme di numeri e dobbiamo poi capire se appartengono all'insieme dei numeri completo.
L'idea di verificatore in tempo polinomiale è la seguente per V:
V = Su input <<S, t>, c>, tale che c è una collezione di numeri (certificato), S è l'insieme di tutti i numeri, c è il certificato:

a. Controlla se c è una collezione di numeri la cui somma è t
b. Controlla se la somma S contenga tutti i numeri
c. Se entrambi passano, *accetta*, altrimenti *rifiuta*

Non sappiamo per certo in quanto tempo polinomiale accada; la selezione stessa sugli insiemi e la determinazione dei due sottoinsiemi verificatori potenzialmente è un problema non deterministico.
Quindi SUBSET-SUM è un problema in NP.

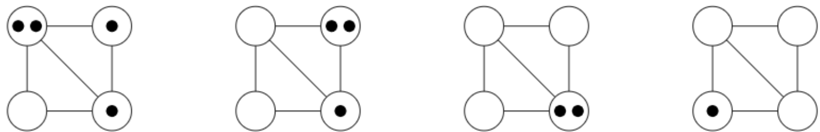
2) Per rappresentare la struttura di SUBSET-SUM/SS come problema NP-Hard, dobbiamo provare che tutti i linguaggi in NP sono riducibili in tempo polinomiale con SETPARTITIONING/SP.
Quindi, abbiamo a che fare:
- per SS un unico insieme tale che la somma sia t
- per SP due insiemi tali che la somma complessiva del primo sia uguale a quella del secondo

Per rappresentare SS prendiamo quindi due insiemi e affermiamo che per entrambi, la somma deve essere uguale a t. Definiamo quindi per l'insieme S_1 un insieme S_{new} a 2t – s, dove s rappresenta la somma degli elementi di S_1. In questo modo, S_1 e S_{new} rappresentano due partizioni che sommano a *t*.
Usando la riduzione, se S_{new} può essere di volta in volta partizionato in una serie di insiemi che contengono 2t – s, significa che avremo sempre che S_{new} è la somma di tutti i numeri interi delle due partizioni che contiene tutti i numeri delle due sottopartizioni.

Quindi, le istanze buone, intese come somme delle sottopartizioni, vengono sommate ad una sola (SS), ma è tale anche che la somma delle due sia spezzabile in due partizioni separate (SS), che è quello che vogliamo realizzare in tempo P.

II/Secondo/2 Appello 2021 – Seconda Parte

3. Pebbling è un solitario giocato su un grafo non orientato G, in cui ogni vertice ha zero o più ciottoli. Una mossa del gioco consiste nel rimuovere due ciottoli da un vertice v e aggiungere un ciottolo ad un vertice u adiacente a v (il vertice v deve avere almeno due ciottoli all'inizio della mossa). Il problema PEBBLEDESTRUCTION chiede, dato un grafo $G = (V, E)$ ed un numero di ciottoli $p(v)$ per ogni vertice v, di determinare se esiste una sequenza di mosse che rimuove tutti i sassolini tranne uno.



Una soluzione in 3 mosse di PEBBLEDESTRUCTION.

Dimostra che PEBBLEDESTRUCTION è NP-hard usando il problema del Circuito Hamiltoniano come problema NP-hard noto (un circuito Hamiltoniano è un ciclo che attraversa ogni vertice di G esattamente una volta).

Per dimostrare che 4DESTRUCTION/PB è NP-Hard, dobbiamo dimostrare che è NP e che è riducibile attraverso l'idea del circuito hamiltoniano.
Descriviamo il certificato per questo problema, tale che si abbia SI in tempo polinomiale.
Esso deve verificare le seguenti condizioni:

- l'insieme dei ciottoli p deve essere collegato ad ogni vertice v
- per ogni vertice, verifica se esiste un arco collegato
- esiste almeno un ciottolo che contiene un sassolino, mentre tutti gli altri non ne contengono nessuno

Dovendo esaminare a coppie i vertici del grafo in PB si ha che la verifica viene eseguita in tempo polinomiale. Per dimostrare che PB è NP-Hard si deve descrivere un algoritmo per risolvere PB attraverso l'uso del Circuito Hamiltoniano/HAMPATH/HM e dimostrare che la riduzione è corretta, tale da trasformare istanze buone di HM in istanze buone di PB e dimostrare che l'istanza è corretta.

Partendo dal circuito hamiltoniano H, sappiamo che esso deve avere tutti i vertici collegati in un ciclo esattamente una volta. Supponiamo quindi che per ogni vertice si abbia almeno un sassolino, tranne un vertice che ne presenta 2, cioè p(v) = 2.
Se abbiamo una riduzione, quindi, l'istanza buona S deve verificare che ogni singolo vertice sia connesso agli altri e che presenti almeno un sassolino; se ciò accade siamo all'interno di un circuito hamiltoniano.
Detto ciò, è possibile che, aggiungendo all'ultimo vertice percorso un sassolino, siamo in un'istanza buona di HM.
Se invece vogliamo verificare se l'istanza S' sia buona per PB, allora prendiamo l'ultimo vertice, che presenta un grado di pebbles/sassolini uguale a 2. Allora è possibile, verificare, togliendo dall'ultimo vertice il sassolino in più e verificando che tutti gli altri vertici contengano un solo sassolino; nel qual caso, abbiamo percorso tutto il ciclo e correttamente abbiamo che si ha un circuito hamiltoniano.
Per verificare la correttezza dell'algoritmo rispetto all'idea iniziale, si ripercorre tutto il circuito, lasciando un solo sassolino all'ultimo vertice e togliendo tutti gli altri con la verifica hamiltoniana.
Se abbiamo che il vertice utilizzato era lo stesso di partenza (indichiamo il vertice u che discutevo prima e v quello attuale dopo l'ultimo ciclo), tale che u = v, allora abbiamo un circuito hamiltoniano che arriva nello stesso punto in cui iniziava il pebble.

Quindi, tutto viene eseguito in tempo polinomiale. Se avessimo anche solo un vertice di troppo o rimanesse un sassolino, correttamente non si avrebbe più un circuito hamiltoniano e neanche il pebble sarebbe risolto. Dunque, PB è un problema NP-Completo e può essere correttamente ridotto ad H.

IV/4/Quarto V/5/Quinto Appello 2021 – Seconda Parte

4. (8 punti) "Colorare" i vertici di un grafo significa assegnare etichette, tradizionalmente chiamate "colori", ai vertici del grafo in modo tale che nessuna coppia di vertici adiacenti condivida lo stesso colore. Considera la seguente variante del problema 4-COLOR. Oltre al grafo G, l'input del problema comprende anche un *colore proibito* f_v per ogni vertice v del grafo. Per esempio, il vertice 1 non può essere rosso, il vertice 2 non può essere verde, e così via. Il problema che dobbiamo risolvere è stabilire se possiamo colorare il grafo G con 4 colori in modo che nessun vertice sia colorato con il colore proibito.

$$\text{CONSTRAINED-4-COLOR} = \{\langle G, f_1, \ldots, f_n \rangle \mid \text{esiste una colorazione } c_1, \ldots, c_n \text{ degli } n \text{ vertici tale che } c_v \neq f_v \text{ per ogni vertice } v\}$$

(a) Dimostra che CONSTRAINED-4-COLOR è un problema NP.
(b) Dimostra che CONSTRAINED-4-COLOR è NP-hard, usando k-COLOR come problema NP-hard di riferimento, per un opportuno valore di k.

a) Per dimostrare che CONSTRAINED-4-COLOR/C4C è in NP deve esistere un verificatore in tempo polinomiale. Tale verificatore, prendendo in input il grafo G, l'insieme dei colori proibiti *f* e l'insieme di tutti i colori *c* verifica che:
- ogni vertice contenga un colore tra i 4 *f*
- verifica che ogni vertice sia collegato ad un vertice con un colore che non sia proibito per lui (come si vede, ai 4 colori accettati, corrispondo 4 colori proibiti), quindi controllando non stia in *c*
- controlla che tutti i vertici siano adiacenti ai colori permessi *(f)* e che le coppie di vertici adiacenti non contengano lo stesso colore (dunque a due a due non stiano in *c*); se tutti i test sono superati accetta, altrimenti rifiuta. Essendo una verifica fatta in tempo polinomiale, C4C sta in NP.
b) Per dimostrare che C4C è NP-Hard, dimostriamo che si può usare C4C come istanza per risolvere k-Color/KC.
La riduzione deve prendere l'insieme dei colori buoni *f* che sta in C4C ed usarli per risolvere KC.
Dimostriamo quindi che:

- sia S un'istanza buona di KC. Allora è possibile associare ad ogni vertice un colore compreso in *f* in tempo polinomiale (in quanto i colori *f* fanno parte di *k*). Per fare ciò consideriamo un grafo G che duplica ogni singolo vertice e forziamo tutti i vertici duplicati ad un colore fisso (che sono i 4 colori di *f*, quindi per k=4). Aggiungendo ogni colore tra i 4 di *f* ad un vertice, otteniamo per certo un grafo in KC e con colori che vanno bene a C4C.
- sia S' un'istanza buon di C4C. Tra tutti i colori di k, sappiamo che esiste il sottoinsieme che va bene (C_1, … C_n) e l'insieme che non va bene (f_1, … , f_m). Dobbiamo verificare a coppie che i vertici adiacenti siano diversi dall'insieme *f*. Dato che ogni coppia di vertici ha colore diverso, verifichiamo che ogni coppia abbia un colore *c* diverso dal corrispondente colore proibito in *f*; siccome abbiamo usato 4 colori, alla fine si riduce ad un controllo polinomiale facendo in modo di verificare che ogni singolo vertice contenga esattamente 4 colori; se ciò avviene, abbiamo un'istanza corretta di C4C.

Il se e solo se si ha dato che, usando 4 colori, di sicuro fanno parte dei k e similmente, partendo da k colori, verifico che siano tutti diversi e ciò avviene avendo usato effettivamente 4 colori diversi in partenza. Pertanto, la riduzione è corretta ed è stata eseguita correttamente in tempo polinomiale.

5. Un circuito Hamiltoniano in un grafo G è un ciclo che attraversa ogni vertice di G esattamente una volta. Stabilire se un grafo contiene un circuito Hamiltoniano è un problema NP-completo.

Un *circuito* 1/3-*Hamiltoniano* in un grafo G è un ciclo che attraversa esattamente una volta *un terzo* dei vertici del grafo. Il *problema del circuito* 1/3-*Hamiltoniano* è il problema di stabilire se un grafo contiene un circuito 1/3-Hamiltoniano.

(a) Dimostrare che il problema del circuito 1/3-Hamiltoniano è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.

(b) Mostrare come si può risolvere il problema del circuito Hamiltoniano usando il problema del circuito 1/3-Hamiltoniano come sottoprocedura.

Per la parte (a) un certificato è una sequenza di n/3 nodi. E' facile (lineare) verificare se un tale sequenza di nodi forma un circuito hamiltoniano o no. Per la parte (b), mappiamo una qualsiasi istanza del problema del circuito Hamiltoniamo che è formata da un grafo G in un'istanza del 1/3-circuito Hamiltoniano, composta da 3 copie di G completamente disgiunte tra loro. E' facile vedere che se il grafo con 3 copie di G ha un circuito Hamiltoniano su 1/3 dei nodi, lo ha su una delle 3 componenti e quindi su G. Se invece il grafo con 3 copie di G non ha un circuito Hamiltoniano su 1/3 dei nodi, allora non c'è circuito Hamiltoniano su G. Quindi abbiamo ridotto il problema del circuito Hamiltoniano a quello del 1/3 Hamiltoniano, dimostrando, assieme al punto (a), che esso è NP completo.

6. Il problema SetPartitioning chiede di stabilire se un insieme di numeri interi S può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Sappiamo che questo problema è NP-completo.

Considerate il seguente problema, che chiameremo SubsetSum: dato un insieme di numeri interi S ed un valore obiettivo t, stabilire se esiste un sottoinsieme $S' \subseteq S$ tale che la somma dei numeri in S' è uguale a t.

(a) Dimostrare che il problema SubsetSum è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.

Il certificato per SubsetSum *è dato dal sottoinsieme S'. Occorre verificare che ogni elemento di S' appartenga anche ad S e che la somma dei numeri contenuti in S' sia uguale a t.*

(b) Mostrare come si può risolvere il problema SetPartitioning usando il problema SubsetSum come sottoprocedura.

Dato un qualsiasi insieme di numeri interi S che è un'istanza di SetPartitioning, *costruiamo in tempo polinomiale un'istanza di* SubsetSum. *L'insieme di numeri interi di input rimane S, mentre il valore obiettivo t è uguale alla metà della somma degli elementi contenuti in S.*

In questo modo, se S' è una soluzione di SubsetSum, *allora la somma dei valori contenuti nell'insieme $S - S'$ sarà pari alla metà della somma degli elementi di S. Quindi ho diviso S in sue sottoinsiemi $S_1 = S'$ e $S_2 = S - S'$ tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2.*
Viceversa, se S_1 e S_2 sono una soluzione di SetPartitioning, *allora la somma dei numeri contenuti in S_1 sarà uguale alla somma dei numeri in S_2, e quindi uguale alla metà della somma dei numeri contenuti in S. Quindi sia S_1 che S_2 sono soluzione di* SubsetSum *per il valore obiettivo t specificato sopra.*

6. Un circuito Hamiltoniano in un grafo G è un ciclo che attraversa ogni vertice di G esattamente una volta. Stabilire se un grafo contiene un circuito Hamiltoniano è un problema NP-completo.

Considerate il seguente problema, che chiameremo HAM375: dato un grafo G con n vertici, trovare un ciclo che attraversa esattamente una volta $n - 375$ vertici del grafo (ossia tutti i vertici di G *tranne* 375).

(a) Dimostrare che il problema HAM375 è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.
Sia n il numero di vertici del grafo che è istanza di HAM375. Un certificato per HAM375 è una sequenza ordinata di $n - 375$ vertici distinti. Occorre tempo polinomiale per verificare se ogni nodo è collegato al successivo e l'ultimo al primo.

(b) Mostrare come si può risolvere il problema del circuito Hamiltoniano usando il problema HAM375 come sottoprocedura.
Dato un qualsiasi grafo G che è un'istanza del problema del circuito Hamiltoniano, costruiamo in tempo polinomiale un nuovo grafo G' che è un'istanza di HAM375, aggiungendo a G 375 vertici isolati (senza archi). Chiaramente G' ha un ciclo che attraversa esattamente una volta $n - 375$ vertici del grafo se e solo se G ha un ciclo Hamiltoniano.

5. "Colorare" i vertici di un grafo significa assegnare etichette, tradizionalmente chiamate "colori", ai vertici del grafo in modo tale che nessuna coppia di vertici adiacenti condivida lo stesso colore. Il problema kCOLOR è il problema di trovare una colorazione di un grafo non orientato usando k colori diversi.

(a) Dimostrare che il problema 4COLOR (colorare un grafo con 4 colori) è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.

(b) Mostrare come si può risolvere il problema 3COLOR (colorare un grafo con 4 colori) usando 4COLOR come sottoprocedura.

(c) Per quali valori di k il problema kCOLOR è NP-completo?
- ☐ Per nessun valore: kCOLOR è un problema in P
- ☐ Per tutti i $k \geq 3$
- ☐ Per tutti i valori di k

Le risposte seguono:

a) se i vertici del grafo sono numerati da 1 a n, allora una sequenza di lunghezza n dei 4 colori disponibili, dove il colore in posizione i della sequenza è associato al vertice i, è un certificato. Per verificare che la colorazione corrispondente al certificato ha risposta SI, basta verificare che il vertice i abbia colore diverso da tutti i vertici a cui è collegato e questa operazione è lineare nella taglia del grafo , visto che basta esaminare ogni arco del grafo 2 volte: una per ciascuno dei 2 vertici collegati dall'arco.

b) Si riduce 3COLOR a 4COLOR come segue: dato un qualsiasi grafo G, istanza di 3COLOR, si aggiunge a G un vertice collegato a tutti i vertici di G. Il grafo G' così ottenuto è un'istanza di 4COLOR e infatti G è colorabile con 3 colori sse G' lo è con 4 colori. Per cui 4COLOR è almeno altrettanto intrattabile di 3COLOR.

c) I problemi kCOLOR con k≥ 3 sono NP-completi.

5. Un circuito Hamiltoniano in un grafo G è un ciclo che attraversa ogni vertice di G esattamente una volta. Stabilire se un grafo contiene un circuito Hamiltoniano è un problema NP-completo.

Un *circuito quasi Hamiltoniano* in un grafo G è un ciclo che attraversa esattamente una volta tutti i vertici del grafo *tranne uno*. Il *problema del circuito quasi Hamiltoniano* è il problema di stabilire se un grafo contiene un circuito quasi Hamiltoniano.

(a) Dimostrare che il problema è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.
Sia n il numero di vertici del grafo che è istanza di QHC. Un certificato per QHC è una sequenza ordinata di n-1 vertici distinti. Occorre tempo polinomiale per verificare se ogni nodo è collegato al successivo e l'ultimo al primo.

(b) Mostrare come si può risolvere il problema del circuito Hamiltoniano usando il problema del circuito quasi Hamiltoniano come sottoprocedura.
Dato un qualsiasi grafo G che è un'istanza di HC, costruiamo in tempo costante un nuovo grafo G' che è un'istanza di QHC, aggiungendo a G un vertice isolato (senza archi). Chiaramente G' ha un QHC sse G ha un HC.

(c) Il problema del circuito quasi Hamiltoniano è un problema NP-completo?
- ☐ No, è un problema in P
- ☐ No, è un problema NP ma non NP-completo
- ☐ Si

Nei precedenti punti abbiamo dimostrato che QHC è NP e che è NP-hard. Quindi è NP-completo.

5. "Colorare" i vertici di un grafo significa assegnare etichette, tradizionalmente chiamate "colori", ai vertici del grafo in modo tale che nessuna coppia di vertici adiacenti condivida lo stesso colore. Il problema kCOLOR è il problema di trovare una colorazione di un grafo non orientato usando k colori diversi.
   (a) Dimostrare che il problema 5COLOR (colorare un grafo con 5 colori) è in NP fornendo un certificato per il *Sì* che si può verificare in tempo polinomiale.
   (b) Mostrare come si può risolvere il problema 3COLOR (colorare un grafo con 3 colori) usando 5COLOR come sottoprocedura.

a) 5COLOR è in NP in quanto esiste un certificato in tempo polinomiale. In pratica:
- per ogni vertice controllo di avere una coppia adiacente
- per ogni coppia di vertici controlla se hanno lo stesso colore
Se accetta entrambe le condizioni, esiste un certificato in tempo polinomiale.

b) Usiamo il problema 3COLOR usando 5COLOR:
Questo è un problema risolvibile in tempo polinomiale, tale che:
- sia S un'istanza buona di 5C. Se abbiamo almeno, per il verificatore, ogni coppia di vertici conterrà un colore tra i possibili 5 esistenti, allora usando 5Color riusciamo a dire che esistono almeno 3 colori contenuti nell'insieme dei 5 possibili. Banalmente, mentre si assegnano i vertici p possibile verificare in tempo lineare che ciascuna coppia si a diversa e conterrà almeno 3 colori diversi
- sia S' un'istanza buon di 3C. Allora questi 3 colori per ciascuna coppia di vertici sono almeno contenuti in un'istanza buona di 5C. Certificando che le coppie sono tutte diverse tra di loro, risolviamo correttamente il problema.

7.30 Let *SET-SPLITTING* $= \{\langle S, C\rangle |$ S is a finite set and $C = \{C_1, \ldots, C_k\}$ is a collection of subsets of S, for some $k > 0$, such that elements of S can be colored *red* or *blue* so that no C_i has all its elements colored with the same color$\}$. Show that *SET-SPLITTING* is NP-complete.

Immaginiamo esista un verificatore in tempo polinomiale. Esso controlla:
- che esista una serie di sottoinsiemi S
- che esista una serie di colori C
- che ciascun elemento di C sia colorato con colori diversi

Ciascuna verifica può essere eseguita in tempo polinomiale.
Descriviamo ora una riduzione polinomiale, trasformando SET-SPLITTING/SS in una subroutine di 3SAT.
Abbiamo infatti in questo caso a disposizione un set di almeno 3 colori, simili ai valori di verità di SAT, nello specifico s1, s2, s3. Per trasformarla in CNF, almeno uno di questi valori dei colori deve essere diverso dagli altri due; pertanto, uno è a *false* e gli altri a *true*.
Dimostriamo la correttezza della riduzione:
- definiamo una istanza buona del problema, prendendo due sottoinsiemi disgiunti con un corrispondente set di valori di verità; esistendo almeno due set, gli elementi differiscono di tutti gli elementi in merito al colore, nello specifico descrivendo valori tutti veri o tutti falsi. Se il partizionamento riguarda correttamente due insiemi tali che ci siano dei valori disgiunti (veri e falsi, come detto, perché i colori devono essere diversi), allora la riduzione è corretta.

Essendo operazioni eseguite in tempo polinomiale, allora SET-SPLITTING è NP-Completo ed NP-Hard.

In the following solitaire game, you are given an $m \times m$ board. On each of its m^2 positions lies either a blue stone, a red stone, or nothing at all. You play by removing stones from the board until each column contains only stones of a single color and each row contains at least one stone. You win if you achieve this objective. Winning may or may not be possible, depending upon the initial configuration. Let $SOLITAIRE = \{\langle G \rangle |\ G$ is a winnable game configuration$\}$. Prove that $SOLITAIRE$ is NP-complete.

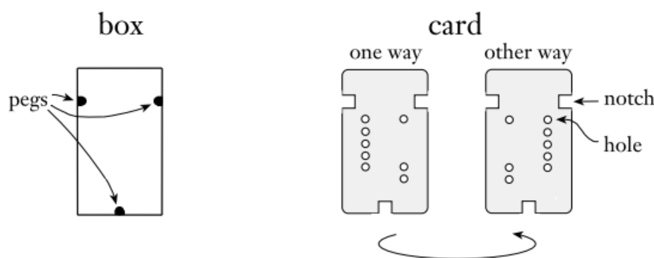
Proviamo SOLITAIRE è NP-Completo.
Dimostriamo che esiste un verificatore

- per ogni riga esiste una pietra specifica
- per ogni colonna esiste una pietra di un singolo colore

Se sono verificate entrambe le condizioni accetta, altrimenti rifiuta.

Per provare che SOLITAIRE è NP-Completo; provo a fare una riduzione con 3SAT.
Per ogni assegnamento delle pietre in riga colonna, quindi (i, j), esiste almeno una riga e colonna che possiede delle pietre e di un singolo colore. Una mossa porta a rimuovere la pietra di un colore almeno di una colonna, che sappiamo essere solo blu o rosse; se rimuoviamo una pietra, almeno un istanza di 3SAT possiede un letterale a vero, quindi vuol dire che rimangono, come altre due condizioni, almeno la pietra di un colore in una riga e dello stesso nella colonna.
Per ogni colonna, si ragiona prendendo *vero* e *falso*; se abbiamo un risultato *vero*, allora vuol dire che in riga e colonna esiste almeno una pietra di un colore e che a sinistra e a destra nella riga ci sono pietre o rosse o blu; se ciò avviene, allora, anche il risultato sarà vero se per una riga esiste una pietra a fianco di un altro colore. Similmente, se controllando un colore (ad esempio blu), mi accorgo che le assegnazioni vicine portano allo stesso colore, allora la condizione di diversità di colore ed unicità della pietra è falsa.
Pertanto, in entrambi versi, la cosa è soddisfatta.
Le istanze cattive del problema sono dato dal non rispetto delle due condizioni di pietra, unicità e colore; le istanze buone portano a *true* e assumono che il problema sia risolvibile.
Essendo due condizioni false e una vera, allora il problema è ridotto correttamente.

You are given a box and a collection of cards as indicated in the following figure. Because of the pegs in the box and the notches in the cards, each card will fit in the box in either of two ways. Each card contains two columns of holes, some of which may not be punched out. The puzzle is solved by placing all the cards in the box so as to completely cover the bottom of the box (i.e., every hole position is blocked by at least one card that has no hole there). Let $PUZZLE = \{\langle c_1, \ldots, c_k \rangle |$ each c_i represents a card and this collection of cards has a solution$\}$. Show that $PUZZLE$ is NP-complete.



Immaginiamo di avere un verificatore che opera in tempo polinomiale. Esso dovrà far rispettare al problema alcune condizioni:
Il verificatore V su input <c, d>, dove “c” è l’insieme delle carte e “d” è l’insieme delle direzioni:

1) piazza le carte in tutte le direzioni

2) controlla per ogni carta se almeno una posizione è bloccata da un buco, verificando quindi che tutta la parte bassa sia completamente coperta

Entrambe le operazioni sono svolte in tempo polinomiale, pertanto il problema è in NP.
Per dimostrare che PUZZLE è NP-Completo, dobbiamo trovare una riduzione polinomiale tale che PUZZLE sia utilizzabile come istanza del problema.
Si può immaginare PUZZLE come sottoproblema di 3-SAT e vogliamo descrivere una riduzione: sapendo che i pezzi del puzzle possono essere combinati in due modi diversi, sappiamo anche che rispettando queste condizioni a due a due, le colonne sono anche vuote e alcune di queste non contengono buchi tali da coprire e riempire correttamente ogni singolo pezzo.
Quindi, avremo un'istanza ϕ che riguarda 3 variabili in maniera tale che una possibile disgiunzione restituisca x_i oppure $\underline{x_i}$ affinché ogni istanza riguardi un accoppiamento a due a due dei buchi tali da coprire tutti i buchi; infatti, dato che possiamo infilare i pezzi in due modi su due colonne di buchi, riempire tutto il fondo riguarda avere in output almeno 3 valori, di cui uno in più descrive lo stato di incastro nel blocco.
La riduzione è corretta dato che:

- sia S un'istanza buona di PUZZLE. Allora, idealmente, ottenendo x_i se stiamo coprendo i buchi nel fondo e altrimenti otterremo $\underline{x_i}$. Dato che possiamo incastrare i pezzi in due modi diversi, il terzo valore (che determina se sia stato incastrato oppure no) garantisce che ci sarà *sempre* almeno un risultato, sia che sia incastrato che no. Infatti, in tempo polinomiale e a forza bruta, possiamo prendere i pezzi e gradualmente incastrarli tutti nelle due colonne. Dunque, avremo un'istanza buona di 3-SAT perché produrremo sempre un valore di verità tale che:
  - avremo *true* se il pezzo si incastra
  - se non riusciamo ad incastrare un pezzo avremo *false*
  - avremo un elemento che contribuisce alla disgiunzione se in una pila di valori, non riusciamo a completare tutti i pezzi di una colonna, dando alternativamente *true* e *false*
- se S invece è un'istanza cattiva, allora non riusciremmo ad incastrare tutti i pezzi; prima o poi, però, si conta in tempo polinomiale di avere un valore *true* che contribuisce all'inizio del processo di incastro di ogni singolo pezzo

La riduzione *f* opera in tempo polinomiale correttamente come subroutine di 3-SAT. Dunque, PUZZLE è NP-Completo.

Let G represent an undirected graph. Also let

$$SPATH = \{\langle G, a, b, k\rangle \mid G \text{ contains a simple path of length at most } k \text{ from } a \text{ to } b\},$$

and

$$LPATH = \{\langle G, a, b, k\rangle \mid G \text{ contains a simple path of length at least } k \text{ from } a \text{ to } b\}.$$

a. Show that $SPATH \in \mathrm{P}$.
b. Show that $LPATH$ is NP-complete.

a) Se SPATH è in P, esisterà un verificatore polinomiale in grado di descriverlo.
Il verificatore V su input <G, a, b, k>:

- controlla che esista un cammino da a verso b di lunghezza k come massima, se ciò non accade rifiuta. Per fare ciò, usa un segno per ogni arco e connette tutto dall'arco i-esimo verso l'arco i+1-esimo
- se tutti i punti sono marcati, allora accetta se ha un valore al più k, avendo contato tutti i mark

Date queste condizioni, verificabili entrambe in tempo polinomiale, allora SPATH è in P

b) Se LPATH è in P, esisterà un verificatore polinomiale in grado di descriverlo.
Il verificatore V su input <G, a, b, k>:

- controlla che esista un cammino non ripetuto nella sequenza di nodi che porta da a verso c
- dato che la lunghezza deve essere al minimo k, allora controlla termine per termine

Date queste condizioni, verificabili entrambe in tempo polinomiale, allora LPATH è in P e, non sapendo per certo se il problema è risolvibile, ma solo verificabile in P, siamo in NP.
Ora vogliamo trovare un problema per ridurre LPATH ad esso, tale che sia NP-Hard.
Un'idea utile viene da HAMPATH, che sarebbe il circuito hamiltoniano.

- Prendiamo l'istanza buona di LPATH e sappiamo che per HAMPATH deve esserci esattamente un solo cammino che collega ogni nodo; pertanto per le proprietà di HAMPATH avremmo che la lunghezza di tutti i vertici meno il primo su cui torniamo è k.
  Dato che percorriamo tutti gli archi con un certo cammino di lunghezza k, tornando sul primo vertice avremo almeno k + 1 nodi, in quanto siamo passati su tutti i vertici almeno una volta.
  Risolviamo quindi HAMPATH
- Similmente, prendiamo una istanza buona di HAMPATH. Se ciò accade, tutti gli archi sono collegati esattamente una volta con tutti i vertici e, facendo ciò, ripercorriamo tutti gli archi tenendo nota della lunghezza di ciascun cammino; è chiaro come avendo un circuito hamiltoniano, la lunghezza dei cammini sarà generalmente data dal numero di vertici meno il primo. Se ciò accade, siamo partiti da un circuito hamiltoniano e si ha che la riduzione è corretta, risolvendo anche LPATH.

La riduzione è quindi corretta, in quanto svolta in tempo polinomiale e vale in entrambi i sensi.
Per questi motivi, LPATH è NP-Completo.

Let $CNF_k = \{\langle\phi\rangle \mid \phi$ is a satisfiable cnf-formula where each variable appears in at most k places$\}$.

a. Show that $CNF_2 \in$ P.
b. Show that CNF_3 is NP-complete.

a) Per mostrare che siamo in P allora dobbiamo mostrare che siamo in CNF; quindi, un letterale è una variabile booleana normale o negata ed ogni frase sia composta da proposizioni di tipo OR e sia composta esattamente da due letterali).
Il verificatore agisce su ϕ in questo modo:

- Verifica che ogni clausola sia composta esattamente da due letterali, siano essi nella forma negata o normale
- Calcola il risultato di ogni clausola e verifica che in ognuna di queste, per il risultato, si abbia un risultato opposto tra una clausola che contiene la negazione di un letterale ed il letterale stesso
- Aggiunge il risultato corretto in questa forma (indica che siamo partiti da OR) e ripete il controllo

Procedendo in questo modo, allora, eseguiamo una scansione lineare e in particolare polinomiale.

b) Per risolvere questo problema, abbiamo bisogno di dimostrare sia NP-Completo e che esista una riduzione accettabile in tempo polinomiale. Decisamente in questo problema di riferimento utilizziamo 3-SAT. E quindi strutturiamo:

- Φ istanza buona di 3-SAT. In questo caso, cerchiamo a due a due la composizione di clausole che contengono x e il suo complementare nelle formule del tipo $(x_1 \vee A_1) \ldots (x_n \vee A_n)$
- Seleziono ciascuna variabile S tale che se abbiamo $(x_1 \vee A_1)$ rimuoviamo dalla formula
- In generale avremmo risultati nella forma:

$$(S_1 \vee A_1) \wedge (\neg S_1 \vee S_2) \wedge (S_2 \vee A_2) \wedge (\neg S_2 \vee S_3) \ldots (S_{\blacksquare} \vee A_{\blacksquare}) \wedge (\neg S_{\blacksquare} \vee S_{\blacksquare})$$

Se abbiamo tutto questo, sappiamo che il controllo di CNF_K, in questo caso CNF_3 è andato a buon fine e la riduzione è corretta.
Partendo invece da una istanza buona di CNF_3, abbiamo che se la formula darà 1 partirà da un letterale e il suo negato per ogni singolo OR, ritornando a clausole della stessa forma iniziale.
Anche qui la riduzione è corretta ed operabile in tempo polinomiale. Per entrambi i motivi, ammettiamo CNF_3 sia NP-Completo e anche CNF_K.

Let $CNF_H = \{\langle\phi\rangle \mid \phi$ is a satisfiable cnf-formula where each clause contains any number of literals, but at most one negated literal$\}$. Show that $CNF_H \in \mathrm{P}$.

Vogliamo mostrare che CNF_H sia in P e quindi vogliamo capire se siamo partiti da una serie di proposizioni utili e che con verificatore V su input ϕ:

- Vuole avere una formula soddisfacibile tale che dia 1 se effettivamente non esiste una negazione di un letterale; se dà 0, allora ce ne sta almeno uno
- Controlla quindi che per ogni clausole, non ci sia più di una ripetizione di un letterale negato; in questo modo si avrà almeno un letterale ed il suo negato ed il calcolo parziale darà 1. Se non succede, si hanno clausole vuote e rifiuta, altrimenti continua a ripetere finché non abbiamo più clausole solitarie del tipo letterale negato
- Dunque, rifiutiamo se in una singola clausola troviamo un solo letterale negato o se appaiono più letterali negati, continuando a verificare in tempo polinomiale

N = "On input $\langle\varphi\rangle$ where φ is a boolean formula in cnf"

1. If φ don't consists a unit clause $(\sim x)$, assume every literals x will be 1 and $(\sim x)$ will be 0 and accept.

2. Repetition performed until these exists no new $(\sim x)$ unit clause:

3. If φ consist a unit clause $(\sim x)$, remove each clauses that contains $(\sim x)$ from φ and remove every occurrences of x from the clause in φ.

4. If an empty clause is exists in φ, reject.

5. Let every literals x in φ be 1, $(\sim x)$ be 0 and accept.

Let ϕ be a 3cnf-formula. An ***≠-assignment*** to the variables of ϕ is one where each clause contains two literals with unequal truth values. In other words, an ≠-assignment satisfies ϕ without assigning three true literals in any clause.

a. Show that the negation of any ≠-assignment to ϕ is also an ≠-assignment.

b. Let $\neq SAT$ be the collection of 3cnf-formulas that have an ≠-assignment. Show that we obtain a polynomial time reduction from $3SAT$ to $\neq SAT$ by replacing each clause c_i

$$(y_1 \vee y_2 \vee y_3)$$

with the two clauses

$$(y_1 \vee y_2 \vee z_i) \quad \text{and} \quad (\overline{z_i} \vee y_3 \vee b),$$

where z_i is a new variable for each clause c_i, and b is a single additional new variable.

c. Conclude that $\neq SAT$ is NP-complete.

a) Per dimostrare che vale la prima cosa, vogliamo essere in una formula booleana e come tale dovrà contenere solo clausole nella forma "v S" oppure "^ S" e, se ciascuna ha un letterale diverso, vale la condizione del valore di verità diverso. Pertanto, anche la negazione letteralmente avrà almeno un letterale ed il suo negato.

b) Partiamo considerando 3SAT, tale da partire da una formula costituita da 2 OR. Per ottenere un input in quella forma, sappiamo che dovremo introdurre, partendo dalle clausole esistenti, delle variabili "b" e di tipo z_i . Semplicemente consideriamo che per ottenere un risultato soddisfacibile 1, il problema parta dal considerare un letterale negato ed un'altra serie di clausole.

Qualora abbiamo un letterale ad 1, poniamo z_i come 0 e, in questo modo, assegniamo 0 anche a b
Così facendo, sappiamo per certo di avere almeno un valore a 0 e la doppia negazione mi assicura, anche grazie alla disgiunzione, di arrivare possibilmente ad un problema soddisfacibile.
Supponendo di essere arrivati ad 1, quindi istanza buona di 3CNF, allora dato che abbiamo una disgiunzione si deve assegnare uno 0 forzando il vincolo dell' ϕ-assegnamento, sulla formula

$(y_1 \vee y_2 \vee z_i)$ and $(\bar{z}_i \vee y_3 \vee b)$ si deve forzare almeno uno 0 per ottenere una doppia disgiunzione ed avere 1.

c) Dato che abbiamo il problema in P, abbiamo una descrizione verificabile in P e la riduzione commentata è valida, allora abbiamo che ≠-SAT è NP-Completo.

If G is an undirected graph, a ***vertex cover*** of G is a subset of the nodes where every edge of G touches one of those nodes. The vertex cover problem asks whether a graph contains a vertex cover of a specified size:

$$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}.$$

Idea:
Per dimostrare che VERTEX-COVER è NP-completo, dobbiamo dimostrare che è in NP e che tutti i problemi NP sono riducibili in tempo polinomiale ad esso. La prima parte è facile; un certificato è semplicemente una copertura di vertice di dimensione k. Per dimostrare la seconda parte, mostriamo che 3SAT è il tempo polinomiale riducibile a VERTEX-COVER. La riduzione converte una formula 3cnf ϕ in un grafo G e un numero k, in modo che ϕ sia soddisfacente ogni volta che G ha una copertura di vertice con nodi k. La conversione avviene senza sapere se ϕ è soddisfacente. In effetti, G simula ϕ. Il grafico contiene gadget che imitano le variabili e le clausole della formula. Progettare questi gadget richiede un po' di ingegno. Per il gadget variabile, cerchiamo una struttura in G che possa partecipare alla copertura del vertice in uno dei due modi possibili, corrispondente alle due possibili assegnazioni di verità alla variabile. Il gadget variabile contiene due nodi collegati da un bordo. Questa struttura funziona perché uno di questi nodi deve apparire nella copertura del vertice. Associamo arbitrariamente TRUE e FALSE a questi due nodi. Per il gadget della clausola, cerchiamo una struttura che induce la copertura del vertice a includere nodi nei gadget variabili corrispondenti ad almeno un valore letterale vero nella clausola. Il gadget contiene tre nodi e spigoli aggiuntivi in modo che qualsiasi copertura di vertice debba includere almeno due dei nodi, o eventualmente tutti e tre. Sarebbero necessari solo due nodi se uno dei nodi gadget variabili aiuta coprendo un bordo, come accadrebbe se il valore letterale associato soddisfacesse tale clausola. In caso contrario, sarebbero necessari tre nodi. Infine, abbiamo scelto k in modo che la copertura del vertice ricercata abbia un nodo per gadget variabile e due nodi per gadget clausola.

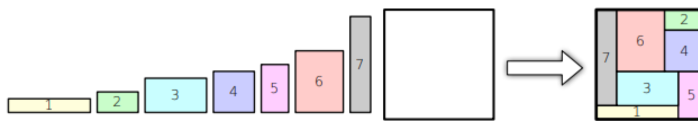
Prova

Ecco i dettagli di una riduzione da 3SAT a VERTEX-COVER che opera in tempo polinomiale. La riduzione mappa una formula booleana ϕ a un grafico G e un valore k. Per ogni variabile x in ϕ, produciamo un bordo che collega due nodi. Etichettiamo i due nodi in questo gadget x e $\underline{x}$. L'impostazione x su TRUE corrisponde alla selezione del nodo etichettato x per la copertura del vertice, mentre FALSE corrisponde al nodo etichettato $\underline{x}$.
I gadget per le clausole sono un po' più complessi. Ogni gadget clausola è un triplo di nodi etichettati con i tre valori letterali della clausola. Questi tre nodi sono collegati tra loro e ai nodi nei gadget variabili che hanno le stesse etichette. Pertanto, il numero totale di nodi che appaiono in G è 2m + 3l, dove ϕ ha m variabili e l clausole. Sia k m + 2l. Ad esempio, se ϕ = (x1 ∨ x1 ∨ x2) ∧ ($\underline{x1}$ ∨ $\underline{x2}$ ∨ $\underline{x2}$) ∧ (x1 ∨ x2 ∨ x2), la riduzione produce <G, k> da ϕ, dove k = 8 e G assume la forma mostrata nella figura seguente.

Per dimostrare che questa riduzione funziona, dobbiamo dimostrare che ϕ è soddisfacente se e solo se G ha una copertura di vertice con nodi k. Iniziamo con un incarico soddisfacente. Per prima cosa inseriamo i nodi dei gadget variabili che corrispondono ai veri valori letterali nell'assegnazione nella copertina del vertice. Quindi, selezioniamo un vero letterale in ogni clausola e inseriamo i restanti due nodi di ogni gadget della clausola nella copertura del vertice. Ora abbiamo un totale di nodi k. Coprono tutti i bordi perché ogni bordo del gadget variabile è chiaramente coperto, tutti e tre i bordi all'interno di ogni gadget della clausola sono coperti e tutti i bordi tra i gadget variabile e clausola sono coperti. Quindi G ha una copertura di vertice con nodi k. In secondo luogo, se G ha una copertura di vertici con k nodi, mostriamo che ϕ è soddisfacente costruendo l'assegnazione soddisfacente. La copertura del vertice deve contenere un nodo in ogni gadget variabile e due in ogni gadget clausola per coprire i bordi dei gadget variabili e i tre bordi all'interno dei gadget clausola. Questo rappresenta tutti i nodi, quindi nessuno è rimasto. Prendiamo i nodi dei gadget variabili che si trovano nella copertina del vertice e assegniamo TRUE ai valori letterali corrispondenti. Tale assegnazione soddisfa ϕ perché ciascuno dei tre bordi che collegano i gadget variabili con ogni gadget clausola è coperto e solo due nodi del gadget clausola sono nella copertura del vertice. Pertanto, uno degli spigoli deve essere coperto da un nodo da un gadget variabile e in modo che l'assegnazione soddisfi la clausola corrispondente.

Il problema SetPartitioning *chiede di stabilire se un insieme di numeri interi S può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Sappiamo che questo problema è NP-hard.*

Il problema RectangleTiling *è definito come segue: dato un rettangolo grande e diversi rettangoli più piccoli, determinare se i rettangoli più piccoli possono essere posizionati all'interno del rettangolo grande senza sovrapposizioni e senza lasciare spazi vuoti.*



Un'istanza positiva di RectangleTiling.

Dimostra che RectangleTiling *è NP-hard, usando* SetPartitioning *come problema di riferimento.*

Idea del prof:

Dimostriamo che RectangleTiling è NP-Hard per riduzione polinomiale da SetPartitioning. La funzione di riduzione polinomiale prende in input un insieme di interi positivi $S = \{s_1, \dots, s_n\}$ e costruisce un'istanza di RectangleTiling come segue:

- i rettangoli piccoli hanno altezza 1 e base uguale ai numeri in S moltiplicati per 3: $(3s_1, 1), \dots, (3s_n, 1)$;
- il rettangolo grande ha altezza 2 e base $\frac{3}{2}N$, dove $N = \sum_{i=1}^{n} s_i$ è la somma dei numeri in S.

Dimostriamo che esiste un modo per suddividere S in due insiemi S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 se e solo se esiste un tiling corretto:

- Supponiamo esista un modo per suddividere S nei due insiemi S_1 e S_2. Posizioniamo i rettangoli che corrispondono ai numeri in S_1 in una fila orizzontale, ed i rettangoli che corrispondono ad S_2 in un'altra fila orizzontale. Le due file hanno altezza 1 e base $\frac{3}{2}N$, quindi formano un tiling corretto.
- Supponiamo che esista un modo per disporre i rettangoli piccoli all'interno del rettangolo grande senza sovrapposizioni né spazi vuoti. Moltiplicare le base dei rettangoli per 3 serve ad impedire che un rettangolo piccolo possa essere disposto in verticale all'interno del rettangolo grande. Quindi il tiling valido è composto da due file di rettangoli disposti in orizzontale. Mettiamo i numeri corrispondenti ai rettangoli in una fila in S_1 e quelli corrispondenti all'altra fila in S_2. La somma dei numeri in S_1 ed S_2 è pari ad $N/2$, e quindi rappresenta una soluzione per SetPartitioning.

Per costruire l'istanza di RectangleTiling basta scorrere una volta l'insieme S, con un costo polinomiale.

Dimostriamo che RectangleTiling/RT è NP-Hard per riduzione polinomiale da SetPartitioning/SP.
L'idea è di usare una funzione di riduzione che prenda in input un insieme di interi positivi
$S = \{s_1, \dots, s_n\}$ e costruisce un'istanza di RectangleTiling seguendo un ragionamento.
Piuttosto che ragionare con la base moltiplicata per 3 per fare in modo di far entrare i rettangoli piccoli senza spazi e sovrapposizioni in quelli grandi, si può ragionare ammettendo che, a prescindere dalla disposizione dei rettangoli, si possa avere: (la moltiplicazione considera base per altezza)

1) Una partizione che dispone i blocchi in orizzontale, che sono idealmente dei blocchi che faremo entrare all'interno di quelli verticali. Ragioniamo per semplicità con multipli di 2, tale che:
   1 x (4*x)|x ∈ S, dunque abbiamo un sottoinsieme di 4 rettangoli che mettiamo in orizzontale.
   L'idea è che poi almeno la metà di questi sia contenuti in due di quelli verticali, moltiplicando i pezzi per due (così uniamo i singoli pezzi senza spazi e assicuriamo che siano divisi equamente).
   La "x" indica un prodotto per tutti i blocchi
2) Una partizione che dispone i blocchi in verticale, tale che:
   2 x $\sum_{x \in S}$ 2x in maniera tale che 4 rettangoli posti in orizzontale siano disposti a due a due in rettangoli verticali.

La logica è simile intuitivamente al ragionamento del prof:

- lui dice che abbiamo dei blocchi che in un sottoinsieme sono di altezza 1 e corrispondono ai numeri moltiplicati per 3 e nell'altro abbiamo blocchi di altezza doppia (2) con una base che è la metà di 2.
  Ciò assicura che noi incastriamo blocchi orizzontali e verticali ragionando su due metà:
  - nel primo abbiamo altezza a metà del secondo
  - nel secondo abbiamo base a metà del primo

L'idea di incastro segue una logica di "compensazione" possiamo dire, anche nell'esempio a 2:

- il primo ha base dimezzata rispetto al secondo, ma altezza doppia rispetto al secondo
- il secondo ha base doppia rispetto al primo, ma altezza dimezzata rispetto al primo

Quindi, dato che la somma di un sottoinsieme S_1 deve corrispondere in S_2, con questa logica, riusciamo correttamente ad "incastrare" i pezzi non assicurando spazi.
I rettangoli piccoli vengono incastrati correttamente a due a due secondo il mio ragionamento e la somma dei numeri di entrambe le partizioni comunque equivale ad N/2, che è una soluzione giusta per SP.
Se ciò avviene, avevamo una istanza buona di RT;

- partendo quindi da due sottoinsiemi di SP, uno ha base doppia ed uno ha altezza doppia. Entrambi convergono ad n/2, incastrando ogni pezzo.

Detto questo, RT è NP-Hard e viene descritto in tempo polinomiale.

BIN PACKING

Instance: Finite set U of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, an integer bin capacity B, and a positive integer K.
Question: Is there a partition of U int disjoint sets $U_1, \dots, U_K$ such that the sum of the sizes of the items inside each U_i is B or less?

(A) Show that the BIN PACKING problem is NP-COMPLETE

Il problema BinPacking/BP richiede che il numero di contenitori imballati non vuoti sia minimo.
Partiamo del SetPartitioning/SP come problema e dimostriamo che esiste per questo una riduzione.
Il problema è simile a quello dei rettangoli, infatti vogliamo idealmente costruire un insieme che abbiamo esattamente il doppio di imballaggi dell'altro.
In questo modo, sappiamo che la somma dei due sottoinsiemi raggiunge N/2 che è una soluzione per SP.
In particolare, sapendo che la somma del primo è uguale alla somma del secondo, essi dovranno quantomeno raggiungere la capacità B, di cui il primo viene dato dalla sottrazione di B – N/4 = N/2.
Due imballaggi soddisfano la condizione se:

$$\sum_{i \in S} c_i = \frac{1}{2} * \sum_{i \notin S} c_i.$$

Se ciò avviene siamo partiti da un'istanza buona del problema ed esaminata in tempo polinomiale (assegnazione degli imballaggi in tempo lineare) e BP è NP-Completo.

MAX SAT

Instance: Set U of variables, a collection C of disjunctive clauses of literals where a literal is a variable or a negated variable in U.
Question: Find an assignment that maximized the number of clauses of C that are being satisfied.

(A) Prove that MAX SAT is NP-Hard.

MAXSAT è NP-Hard e per dimostrarlo usiamo SAT.
Avendo come certificato del problema un assegnamento di verità delle variabili possibili, se per ogni circuito esiste un'assegnazione, si ha una verifica di ciascuna porta logica in tempo polinomiale e il problema è in NP.
Attuiamo una riduzione soddisfacibile il problema:

- sia S istanza buona di SAT. Sapendo che MAX SAT usa un assegnamento delle variabili che considera ogni singola porta logica, l'idea è di usare con un AND logico tutte le singole porte logiche che vanno ad 1, in maniera tale da prendere tutte le clausole che il problema di partenza possiede, arrivando già ad un numero *k* di clausole ad 1. In questo modo, prendiamo tutte le clausole utili a SAT, garantendo ce ne siano almeno *k*.
- Avendo S' istanza buona di MAXSAT, sappiamo che prenderemo almeno un numero di clausole *k* tali da soddisfare il problema precedente.

L'analisi intercorre in tempo polinomiale. Per queste considerazioni, MAXSAT è NP-Completo.

Consider the following scheduling problem. You are given a list of final exams $F_1, \ldots, F_k$ to be scheduled, and a list of students $S_1, \ldots, S_l$. Each student is taking some specified subset of these exams. You must schedule these exams into slots so that no student is required to take two exams in the same slot. The problem is to determine if such a schedule exists that uses only h slots. Formulate this problem as a language and show that this language is NP-complete.

Per formulare il problema come NP-Completo, immaginiamo esista un verificatore in tempo polinomiale V che prende in input un insieme di esami F $\{F_1, \ldots, F_n\}$ e un insieme di studenti S $\{S_1, \ldots, S_n\}$ tale che:

- Verifica che ogni studente sia associato ad un solo esame; se vede che uno studente appartiene a due time slot, rifiuta
- Verifica che uno schedule abbia esattamente *h* slot, per fare ciò deve essere verificato il primo punto e sappiamo che se esistono assegnazione univoca, similmente varrà anche questo.
  Se non è di *h* slot, rifiuta

Valenti entrambe le condizioni, verificabili in tempo polinomiale, il linguaggio è correttamente in NP.
Si può usare il circuito Hamiltoniano per ridurre il problema ad un'assegnazione univoca di vertici, tale che per ogni vertice esista esattamente un arco che lo collega agli altri, formando un ciclo; ad ogni coppia di vertici adiacenti corrisponde l'assegnazione studente-esame e, una volta percorso tutto, sappiamo che sarà di lunghezza *h* e quindi siamo in presenza di un problema ridotto correttamente da HAMILTON.
Similmente, è facile vedere che se abbiamo un'occorrenza giusto di StudentScheduling, allora siamo partiti da un grafo hamiltoniano. A queste condizioni, il problema è NP-Completo.

Fornisci un verificatore polinomiale per il seguente problema:

$$\text{DOUBLEHAMCIRCUIT} = \{\langle G\rangle \mid G \text{ è un grafo non orientato che contiene un ciclo che visita ogni vertice } \textit{esattamente due volte} \text{ e attraversa ogni arco } \textit{esattamente una volta}\}$$

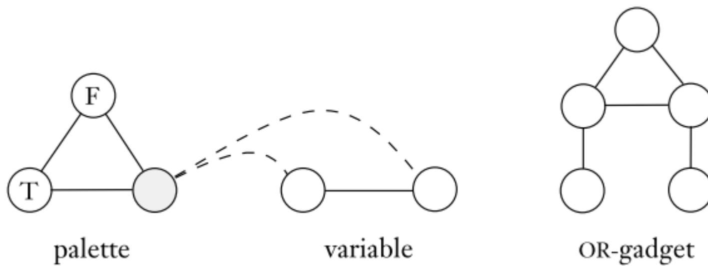
V = "su input $\langle G, C\rangle$, dove G è un grafo ed il certificato C è una sequenza di vertici $(v_1, \ldots, v_k)$:

1. Controlla che ogni elemento di C sia un vertice del grafo;
2. controlla che C sia un ciclo $(v_1 = v_k)$;
3. controlla che ogni vertice del grafo compaia 2 volte in C, tranne v_1 che deve comparire 3 volte;
4. controlla che (v_i, v_{i+1}) sia un arco del grafo per ogni $i = 1, \ldots, k-1$;
5. controlla che ogni arco compaia in C esattamente una volta;
6. se tutti i test sono superati accetta, altrimenti rifiuta."

A ***coloring*** of a graph is an assignment of colors to its nodes so that no two adjacent nodes are assigned the same color. Let

$$3COLOR = \{\langle G\rangle \mid G \text{ is colorable with 3 colors}\}.$$

Show that *3COLOR* is NP-complete. (Hint: Use the following three subgraphs.)



L'idea è di mostrare che 3COLOR sia in NP.
Per mostrare che un grafo sia colorabile con 3 colori, allora deve esistere un verificatore V che con input G

- Controlla che per ogni coppia di vertici adiacenti vi sia una palette (quindi almeno altri 3 nodi) collegati ad ogni coppia, se ciò non accade rifiuta
- Controlla che l'assegnazione abbia come predicato di verità Vero/Falso e una base comune; se ciò non accade, il grafo presenta almeno un colore doppio e rifiuta
- Controlla che per ogni vertice vi sia almeno un arco che lo collega ad altri due vertici a triangolo; se ciò non accade, rifiuta

L'idea è quindi di creare un grafo tale che per ogni variabile x, siano collegate due nodi v_i e $\underline{v_i}$ tale che se hanno lo stesso colore si abbia Vero; pertanto si deve avere un'istanza di SAT falsa a coppie.
Usiamo quindi 3COLOR come istanza di 3-SAT:

- Per ogni nodo, lo consideriamo e ne marchiamo altri due, quindi assegnando tre etichette A,B,C
- Per ogni clausola composta da queste tre etichette, avranno una base comune se abbiamo *false*; quindi, siamo partiti da due vertici collegati con un colore diverso. In particolare, avremo un triangolo formato da tre valori, *True, False, Base* per A,B,C e aggiungendo un gadget con un OR (quindi un gadget conterrà altre 3 etichette)
- Siccome abbiamo 3 vertici, si ha che almeno uno di questi deve essere a vero e gli altri possono essere a vero e a falso, perché consideriamo che partiamo già da un vertice che ha colore diverso da tutti gli altri. Siccome l'istanza deve essere soddisfacibile, allora, è possibile certamente collegarsi con una clausola che ha *True* come output, in quanto 3-Color assicura questa condizione.

Se abbiamo un'istanza buona di 3-SAT, si può dire che il vertice di partenza sia *True* e dunque siamo partiti da un'istanza buona che ha assegnato tramite gadget 3 colori diversi. Non possiamo avere 3 colori a false, perché altrimenti il gadget sarebbe False; il gadget deve essere True perché viene sempre collegato ad una

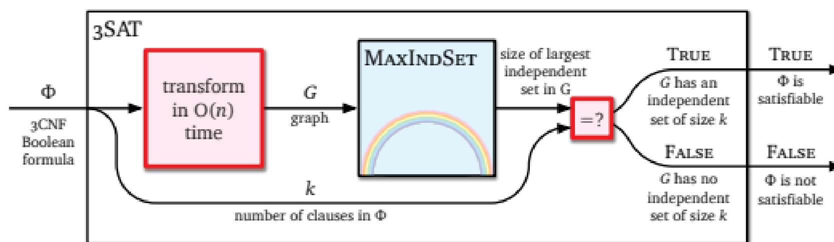
base, che di per sé *deve* essere false, tale da collegare almeno altri due vertici True. Quindi 3-COLOR è NP-Completo ed NP-Hard perché correttamente ridotto da 3-SAT.

Concludiamo risolvendo i vari problemi NP-Completi presenti nelle slide:

MaxIndSet = Sia G = (V, E) un grafo non orientato. Un insieme indipendente in G è un sottoinsieme I dei vertici tali che per ogni coppia di vertici in I, non c'è nessun arco che li collega.
Input: un grafo non orientato G = (V, E)
Output: la dimensione k dell'insieme indipendente più grande in G.

L'idea è di usare una riduzione con SAT, in particolare sapendo che nessun arco collega una coppia di vertici. In particolare, avendo 3 vertici, si usa 3-SAT.

Dimostriamo che **MaxIndSet** è NP-hard con una riduzione da **3SAT**



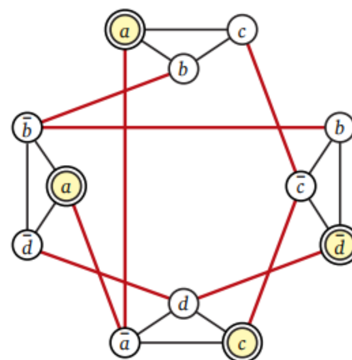
Il verificatore opera in questo modo:

- Prende un grafo G e controlla che ogni coppia di vertici non sia collegata da un arco; se ciò accade rifiuta
- Controlla che per ogni vertice sia collegato almeno 1 un vertice adiacente e a questo ne siano collegati 2, per un totale di 3; se ciò non accade *rifiuta*

Le operazioni sono svolgibili in tempo polinomiale. MaxIndSet è in NP.

Il verificatore dà già in qualche modo l'idea che va seguita. Dobbiamo risolvere 3-SAT usando MaxIndSet come subroutine. In particolare, riduciamo 3-SAT a MaxIndSet.
Quindi partiamo da una formula in 3CNF, in cui costruiamo un grafo con 3k vertici, uno per ogni letterale.
I due vertici sono connessi da un arco se e solo se fanno parte della stessa clausola. In questo caso, se fanno parte della stessa clausola, dovranno dare *true*, altrimenti esistono archi di consistenza, che collegano un letterale alla sua negazione.
Un esempio di formula è:

$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{c})$$



Attraverso 3CNF usiamo 4 clausole perché il set indipendente ha un set con *al più* k nodi indipendenti.
Ogni insieme indipendente in G contiene al massimo un vertice da ogni clausola triangolare, perché due vertici qualsiasi in ogni triangolo sono collegati.
Quindi appunto, il più grande insieme indipendente in G ha dimensioni al massimo k.

Questo discorso è utile perché noi *consideriamo già che quello che ammettiamo sia soddisfacibile*, basta porre i termini giusti del problema. In questo senso, il massimo insieme indipendente è formato certamente da una clausola con tutti i nodi = *true*, pertanto due vertici collegati da un arco non possono esistere. Essendo un "se e solo se" si dimostra in due sensi:

- Supponendo che 3-SAT sia soddisfacibile, allora tutte le clausole restituiscono *true* in virtù del fatto che ci sono i vertici a due a due collegati e con al più un solo vertice in mezzo.
  Per definizione, ogni clausola contiene almeno un valore letterale a *true*. Quindi, possiamo scegliere un sottoinsieme S di k vertici in G che contiene esattamente un vertice per triangolo di proposizione, tale che i corrispondenti k letterali sono tutti *true*. Poiché ogni triangolo contiene al massimo un vertice in S, non ci sono due vertici in S collegati da uno arco triangolare (quindi una clausola di tre a false). Poiché ogni valore letterale corrispondente a un vertice in S è *vero*, non ci sono due vertici in S collegati da uno arco tale da formare un triangolo. Dao che ogni letterale è corrispondente a un vertice in S, non ci sono due vertici in S collegati da un arco con una negazione. Quindi, S in un insieme indipendente di dimensione *k* in G.
- Supponendo invece di avere un'istanza buona di MaxIndSet, sappiamo essere valida in quanto ogni clausola triangolare è formata da *true* se e solo se sussiste la condizione del problema. Ad ogni letterale possono essere assegnate variabili di ogni tipo. Siccome S contiene un vertice in ogni clausola triangolare, ciascuna clausola nella formula, contiene almeno un letterale a *true*

La trasformazione è in tempo polinomiale in modalità *brute-force*. Concludiamo che MaxIndSet è NP-Hard.

Data una formula arbitraria in 3CNF, come per esempio

$$(a \vee b \vee c) \wedge (b \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee c \vee d) \wedge (a \vee \overline{b} \vee \overline{c})$$

costruiamo un grafo $G = (V, E)$ tale che:

- V contiene 3 vertici per clausola, 1 per letterale
- Gli archi in E sono di due tipi
- Archi di clausola che collegano letterali nella stessa clausola
- Archi di consistenza che collegano un letterale con la sua negazione

La correttezza della riduzione:

- Se k è il numero di clausole nella formula, allora un insieme indipendente in G può contenere al più k elementi.
- Dimostriamo che G contiene un insieme indipendente di dimensione esattamente k se e solo se la formula Φ è soddisfacibile:
  - $\Rightarrow$ Se la formula è soddisfacibile allora esiste un assegnamento delle variabili che la rende vera. Scegliamo un sottoinsieme S di k vertici di G, uno per ogni clausola, in modo che il letterale corrispondete sia vero. Si può far vedere che S è un insieme indipendente per G.
  - $\Leftarrow$ Supponiamo che G contenga un insieme indipendente S di dimensione k. Ogni vertice di S deve stare in un triangolo diverso. Se assegnamo il valore Vero ai letterali presenti in S otteniamo un assegnamento che rende vera la formula.

Un problema apparentemente simile è *MaxMatch*.

- Sia $G = (V, E)$ un grafo arbitrario.
- Un accoppiamento o insieme di archi indipendenti in G è un sottoinsieme M degli archi tali che non c'è nessun vertice in comune tra due archi.

Problema del Massimo Accoppiamento (MaxMatch)

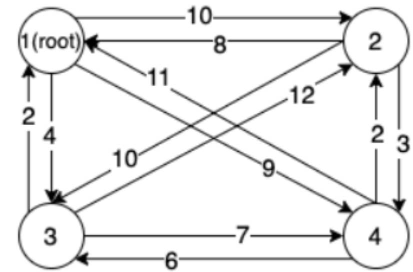
Input: un grafo arbitrario G

Output: la dimensione k dell'accoppiamento più grande in G

Algoritmo di Edmonds

Il problema dell'accoppiamento massimo è risolvibile in tempo polinomiale! Più precisamente, in tempo $O(|V|^2|E|)$

L'algoritmo funziona ricorsivamente e considera un grafo completamente connesso.

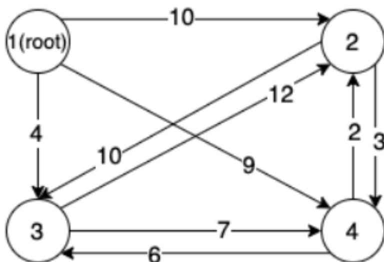


Passo zero:

Dato il grafico mostrato sopra, il primo passo è decidere il punto di partenza, cioè la radice dell'albero. Può essere predefinito dall'utente, altrimenti la radice è il nodo con la più alta somma di bordi in uscita,

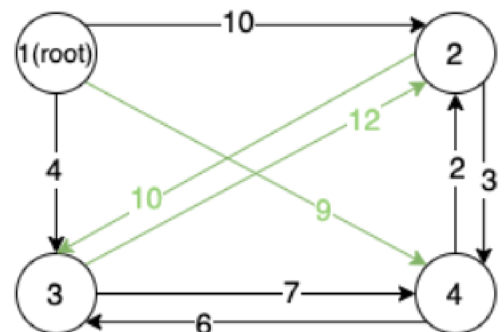
$r = \arg\max_{v_i \in V} \sum_{v_j \in V} e_{ji}$

Dato che la radice può avere solo i bordi in uscita, rimuoviamo tutti i bordi in entrata della radice.
In questo caso, la radice è il nodo 1 e dopo aver rimosso i bordi non necessari, il grafico risultante viene mostrato come:



Passo uno

Innanzitutto, partiamo dal grafico MG con il bordo massimo in entrata per ogni nodo (diverso dalla radice), cioè per ogni nodo vi c'è un solo bordo in entrata dal nodo π(vi) che denota eπ(vi) , ed è il bordo con il peso massimo. Se il grafico è un albero, allora è l'albero di copertura massima (maximum spanning/maximum match), altrimenti il grafico contiene almeno un ciclo, quindi dobbiamo rompere i ciclo sostituendo determinati archi con archi al di fuori del grafico MG.
Tornando all'esempio, i bordi verdi nel grafico seguente formano il grafico MG e possiamo trovare un cerchio tra il nodo 2 e il nodo 3.



Passo due (ricorsivo)

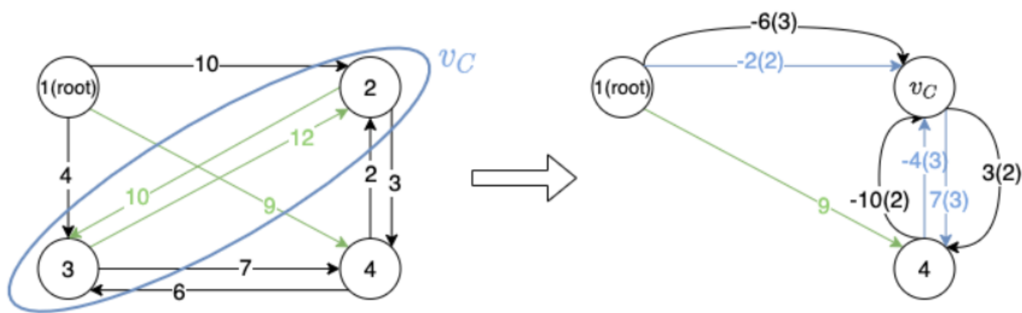
Scegliendo casualmente un ciclo C_{node}={vc 1,v c2,... vck} in MG, il ciclo stesso è ottimale e dobbiamo solo rompere il ciclo con il costo minimo. Per raggiungere questo obiettivo, costruiamo prima un nuovo grafico G' trattando il cerchio come un nuovo nodo vC; quindi, troviamo l'albero di spanning massimo A nel nuovo grafico G' (eseguire ricorsivamente l'algoritmo su G').

Now we first decribe the way to build new graph $G'' = \{V', E''\}$ with the vertex set $V' = V \setminus C \cup \{v_C\}$. As for the edges E', we split them into three cases:

1. For edge e_{sd} in E, if $s \notin C_{node}$ and $d \in C_{node}$, then we add an edge e'_{sv_C} to E' with weight $w(e'_{sv_C}) = w(e_{sd}) - w(e_{\pi(v_d)v_d})$ (it is a negative value)
2. For edge e_{sd} in E, if $s \in C_{node}$ and $d \notin C_{node}$, then we add an edge $e'_{v_C d}$ to E' with weight $w(e'_{v_C d}) = w(e_{sd})$
3. For edge e_{sd} in E, if $s \notin C_{node}$ and $d \notin C_{node}$, then we add an edge e'_{sd} to E' with weight $w(e'_{sd}) = w(e_{sd})$

Then there might be multiple edges between v_C and other nodes, we only keep the edge with maximum weight between v_C and each other node.

In this example, there is only one circle formed by node 2 and node 3, so we treat the circle as a new node v_C, as shown in the left figure below. Then by applying the rules mentioned above, we build the new graph G' shown in the left. Apparently there are multiple edges between node $1, 4$ and node v_C, then we only keep the one with the highest weight (i.e. the blue edges, and the number in the parenthesis represents the original source/destinition of the edge in the circle).

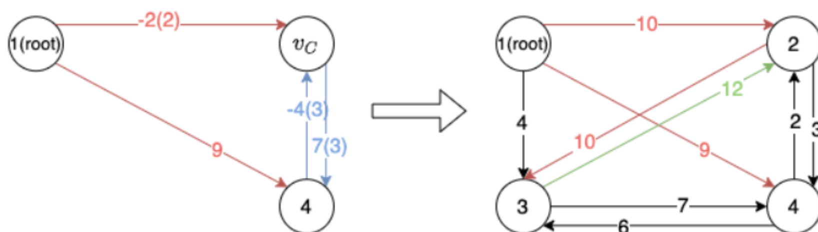


Then we need to find the maximum spanning tree A for the new graph G', and it is formed by the edges in red in the figure below.

Step Three

Without loss of generality, assume the incoming edge of v_C in A is from node v_s and its corresponding edge in the original graph G is e_{sk}, with $v_k \in C_{node}$, then the edges of the final tree is formed by the combination of edges in A (replacing the edges from/to v_C to the original edge) and the edges in the circle without the incoming edge of node v_k.

As shown in the graph below, E^t is formed by the red edges in the right figure.

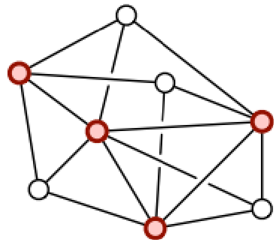


We got it! Congrats! ^.^

L'idea di massima quindi è: partire dalla radice e avere la stessa che possiede tutti i nodi. Se ciò avviene, correttamente dato che siamo partiti da lì, non ci sono vertici comuni se non nella radice.
Quindi si può usare SAT inteso come valore di verità.

Altro problema: *MinVertexCover*.

- Sia $G = (V, E)$ un grafo non orientato.
- Una copertura tramite vertici (Vertex Cover) in G è un sottoinsieme C dei vertici tali che ogni arco ha almeno un'estremità in C



Problema del Minimum Vertex Cover (MinVertexCover)

Input: un grafo non orientato $G = (V, E)$

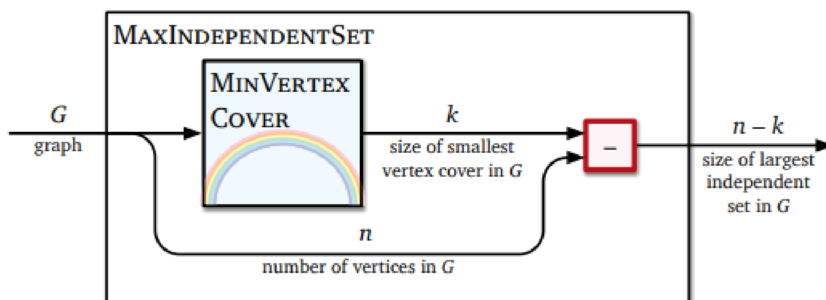
Output: la dimensione k della più piccola copertura tramite vertici di G

Dimostriamo che **MinVertexCover** è NP-hard con una riduzione da **MaxIndSet**

Fatto

I è un insieme indipendente di G, se e solo se $V \setminus I$ è una copertura tramite vertici di G.

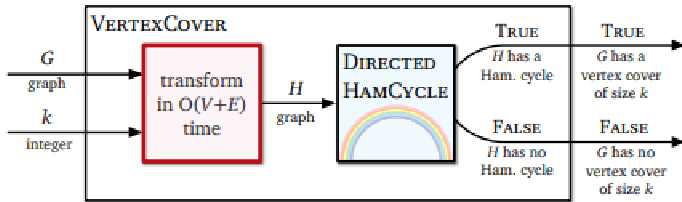
La dimostrazione che *MinVertexCover* è ancora più semplice, perché si basa sulla seguente facile osservazione: I è un insieme indipendente in un grafo G = (V, E) se e solo se il suo complemento V \ I è una copertura di vertice dello stesso grafico G. Quindi, il più grande insieme indipendente in qualsiasi grafico è il complemento della *più piccola* copertura di vertice t dello stesso grafico! Quindi, se la più piccola copertura di vertice in un grafo n-vertice ha dimensione k, allora il più grande insieme indipendente ha dimensione n – k.



Ora abbiamo un problema più grafico: *DirectedHamCycle*.

Dato un grafo G, un **Circuito Hamiltoniano** è un ciclo nel grafo che attraversa tutti i vertici una sola volta.

Dimostriamo che **DirectedHamCycle** è NP-hard con una riduzione da **VertexCover**

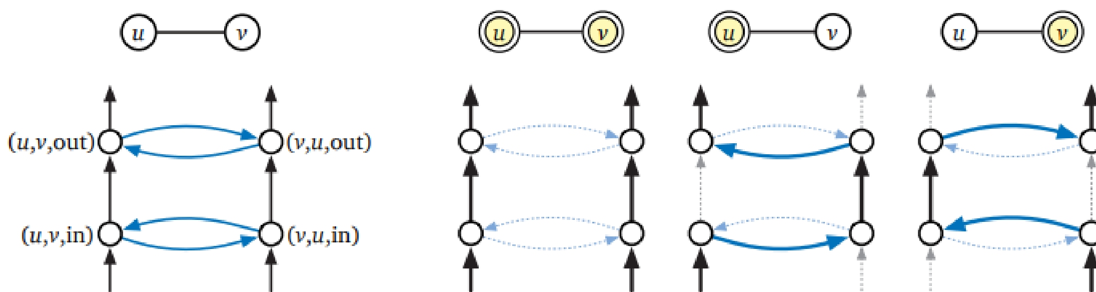


Per ogni spigolo non diretto uv in G, il grafo diretto H contiene un gadget spigolo costituito da quattro vertici (u, v, in), (u, v, out), (v, u, in), (v, u, out) e sei spigoli diretti (u, v, in)

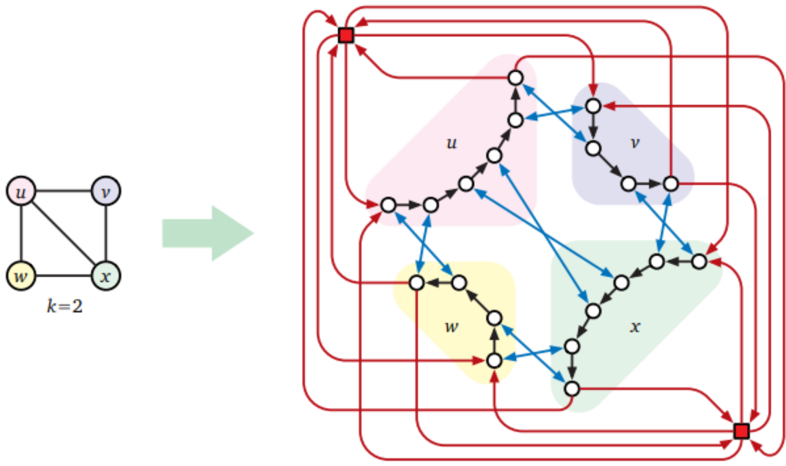
$$(u,v,\text{in})\to(u,v,\text{out}) \qquad (u,v,\text{in})\to(v,u,\text{in}) \qquad (v,u,\text{in})\to(u,v,\text{in})$$
$$(v,u,\text{in})\to(v,u,\text{out}) \qquad (u,v,\text{out})\to(v,u,\text{out}) \qquad (v,u,\text{out})\to(u,v,\text{out})$$

Ogni vertice "in" ha uno spigolo in entrata aggiuntivo e ogni vertice "out" ha uno spigolo in uscita aggiuntivo. Qualsiasi ciclo hamiltoniano in H deve passare attraverso un gadget bordo in uno dei tre modi: dritto su entrambi i lati o con una deviazione da un lato all'altro e viceversa. Alla fine, queste opzioni corrisponderanno sia a u che a v, solo u o solo v appartenenti a qualche copertura di vertice.

Per ogni vertice u in G, tutti i gadget di spigolo per gli spigoli incidenti uv sono collegati in H in un unico percorso diretto, che chiamiamo catena di vertici. In particolare, supponiamo che il vertice u abbia d vicini v1, v2,..., vd . Allora H ha d – 1 spigoli aggiuntivi (u, v_i, out) → (u, v_{i+1}, in) per ogni i da 1 a d – 1.



Infine, H contiene anche k vertici di copertura x1, x2,..., xk. Ogni vertice di copertura ha uno spigolo diretto al primo vertice in ogni catena di vertici e uno spigolo diretto dall'ultimo vertice in ogni catena di vertici



Innanzitutto, supponiamo che C = {u1, u2,..., uk} sia una copertura di vertice di G di dimensione k. Possiamo costruire un ciclo hamiltoniano in H che "codifica" C come segue. Per ogni indice i da 1 a k, attraversiamo un percorso dal vertice di copertura xi, attraverso la catena di vertici per ui, per coprire il vertice xi + 1 (o coprire il vertice x1 se i = k). Mentre attraversiamo la catena per ogni vertice ui, determiniamo come procedere da ciascun nodo (ui, v, in) come segue: –

1. Se v appartiene a C, seguire il bordo (ui, v, in)→ (ui, v, out)
2. Se v non appartiene a C, deviare attraverso (ui, v,in)→ (ui, v, out) → (v, ui, out) → (ui, v, out)

Quindi, per ogni spigolo uv di G, il ciclo hamiltoniano visita (u, v,in) e (u, v, out) come parte della catena dei vertici di ui se si 2 C e come parte della catena dei vertici di v altrimenti.

D'altra parte, supponiamo che H contenga un ciclo hamiltoniano C. Questo ciclo deve contenere uno spigolo da ogni vertice di copertura all'inizio di una catena di vertici. La nostra analisi del caso dei gadget di spigolo implica induttivamente che dopo che C entra nella catena dei vertici per un certo vertice u, deve attraversare tutta la catena di vertici. In particolare, ad ogni vertice (u, v,in), il ciclo deve contenere il singolo spigolo (ui, v, in) (ui, v, out)→ il percorso di deviazione.
(ui, v,in) → (ui, v, out) → (v, ui, out) → (ui, v, out) , seguito da un gadget edge al bordo successivo nella catena di vertici di u o a un vertice di copertura se questo è l'ultimo gadget di spigolo nella catena di vertici di u. In particolare, se C contiene lo spigolo di deviazione (u, v, in) → (v, u, in) non può contenere spigoli tra qualsiasi vertice di copertura e la catena di vertici di v. Ne consegue che C attraversa esattamente k catene di vertici. Inoltre, queste catene di vertici descrivono una copertura di vertice del grafo originale G, perché C visita il vertice (u, v,in) per ogni spigolo uv in G.

0-1 Integer Programming

0-1 INTEGER PROGRAMMING
Instance: A set of linear inequalities over Boolean variables $x_1, x_2, \ldots x_n$ with rational coefficients.
Answer: "Yes" if there is an assignment of values in $\{0, 1\}$ that satisifies the inequalities.

Theorem

0-1 INTEGER PROGRAMMING *is* **NP**-*complete.*

Proof.

Clearly 0-1 INTEGER PROGRAMMING is in **NP**. We reduce 3-SAT to it. For each i add inequalities $0 \le x_i \le 1$. This ensures the integer values are 0 or 1. Second, express each clause as an inequality. E.g., express $(x_1 \vee \bar{x_2} \vee x_3)$ as $x_1 + (1 - x_2) + x_3 \ge 1$. □

Showing the problem is NP-hard

We will show 0-1 integer programming is NP hard by showing:

$$\text{3-CNF-SAT} \leq_{\text{P}} \text{0-1-Integer-Programming}$$

Reduction

For 3-CNF formula Φ, containing v variables and c clauses, construct $c \times v$ matrix A such that:

$$a_{i,j} = \begin{cases} -1 & \text{if variable } j \text{ occurs only without negation in clause } i \\ 1 & \text{if variable } j \text{ occurs only with negation in clause } i \\ 0 & \text{otherwise} \end{cases}$$

Also, construct c-vector b such that:

$$b_i = -1 + \sum_{j=1}^{v} \max(0, a_{i,j})$$

In other words: $b_i = -1$ plus the number of negated literals in clause i.

We observe that A and c can be constructed in $\mathcal{O}(cl)$, proving that the reduction can be done in polynomial time.

We will now prove that the existence of c-vector $x : Ax \leq b \iff \Phi$ is satisfiable. Consider vector x to represent an assignment of the variables in Φ, where:

$$x_i = \begin{cases} 1 & \text{if variable } i \text{ is assigned True} \\ 0 & \text{if variable } i \text{ is assigned False} \end{cases}$$

Let $y = Ax$, then:

$$y_i \leq b_i \iff \text{sum of satisfied literals in clause } i \geq 1 \iff \text{clause } i \text{ is satisfied}$$

From this equation:

$$y = Ax \leq b \iff x \text{ is an assignment satisfying } \Phi$$

Thus we have expressed a 3-CNF-SAT problem as a 0-1 integer programming problem. ■

Traveling Salesman Problem

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.

The traveling salesman problem can be described as follows:
TSP = $\{(G, f, t)$: $G = (V, E)$ a complete graph,
f is a function $V \times V \rightarrow Z$,
$t \in Z$,
G is a graph that contains a traveling salesman tour with cost that does not exceed $t\}$.

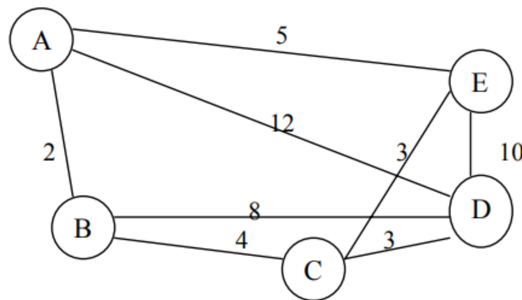


Figure 10.1 A graph with weights on its edges.

The problem lies in finding a minimal path passing from all vertices once. For example the path Path1 {A, B, C, D, E, A} and the path Path2 {A, B, C, E, D, A} pass all the vertices but Path1 has a total length of 24 and Path2 has a total length of 31.

L'idea portante è una riduzione tramite il circuito hamiltoniano. Dato che operiamo una riduzione con esso, attraversiamo tutti i vertici solo una volta e la somma di ciascuno è minima.

Proof:
First, we have to prove that TSP belongs to NP. If we want to check a tour for credibility, we check that the tour contains each vertex once. Then we sum the total cost of the edges and finally we check if the cost is minimum. This can be completed in polynomial time thus TSP belongs to NP.
Secondly we prove that TSP is NP-hard. One way to prove this is to show that Hamiltonian cycle $\leq_P$ TSP (given that the Hamiltonian cycle problem is NP-complete). Assume $G = (V, E)$ to be an instance of Hamiltonian cycle. An instance of TSP is then constructed. We create the complete graph $G' = (V, E')$, where $E' = \{(i, j): i, j \in V$ and $i \neq j$. Thus, the cost function is defined as:

$$t(i, j) = \begin{cases} 0 \text{ if } (i, j) \in E, \\ 1 \text{ if } (i, j) \notin E. \end{cases} \qquad 10.1$$

Now suppose that a Hamiltonian cycle h exists in G. It is clear that the cost of each edge in h is 0 in G' as each edge belongs to E. Therefore, h has a cost of 0 in G'. Thus, if graph G has a Hamiltonian cycle then graph G' has a tour of 0 cost.
Conversely, we assume that G' has a tour h' of cost at most 0. The cost of edges in E' are 0 and 1 by definition. So each edge must have a cost of 0 as the cost of h' is 0. We conclude that h' contains only edges in E.
So we have proven that G has a Hamiltonian cycle if and only if G' has a tour of cost at most 0. Thus TSP is NP-complete.

Esercizi 19/20 Preparazione esame; soluzione alternativa


```
/*Esercizio 19 file preparazione esame completo*/
R = "Su input <G>, la codifica di un CFG G:"
1 - Applica algoritmo per trasformare in Forma Normale di Chomsky a <G>, ottengo nuova codifica <G'> di CFG
    equivalente a G (che genera tutte e sole le stringhe generabili con G)
2 - Definisco TM M che su input <G>, verifica se esiste regola di G del tipo S → epsilon con S variabile iniziale,
    se tale regola esiste accetta, altrimenti rifiuta
3 - Restituisci il risultato dell'esecuzione di M su input codifica di G', vedi 1


/***Esercizio 20**/
R = "Su input <R,S>, la codifica delle espressioni regolari:"
1 - Costriusci i NFA Ar e As, che sono equivalenti alle espressioni R e S
2 - Attraverso le regole di costruzione dell'automa, crea l'epsilon-NFA Ai che riconosce il linguaggio L(R) intersecato L(S);
 (lo posso fare perché i linguaggi regolari sono chiusi per intersezione)
    Per differenza simmetrica costruisci il DFA D che verifica l'uguaglianza L(Ai) = L(R)
3 - Esegui T la TM che decide E_DFA con input D, se T accetta accetta, altrimenti rifiuta


L(R) C L(S) => L(R) ^ L(S) = L(R)
i casi sono due: L(R) = L(S) (sottoinsieme improprio)
L(R) =/= L(S) (sottoinseme proprio)
```