ADVANCED ALGORITHMS SIMPLE (FOR REAL)

GABRIEL ROVESTI

1 Table of Contents

Disclaimer

All notes were organized based on the 2022/2023 lessons (considering they are almost identical, also the ones from this year 2023/2024 to following I think there will be minimal changes in the future) and all existing notes files. I tried my best with each, providing precision and refinements over contents, just to give you the clearest idea possible over everything.

Also, the idea of this file is to have a complete reference with which to follow the entire course, even listening, in classes and to use it as a more in-depth approach to the whole thing. This follows an historical order, so you will find subsections dedicated to exercises in the same chapter they were originally intended inside this year's lessons.

Basically, formatting and everything is basically copied from lessons, rework of existing notes files, the CLRS book of Algorithms and other online resources, presented to you in a complete way – to make it simple, possibly for real. All of the exercises from classes are present and complete explanations are offered, often with examples, to be absolutely complete.

The professor is really good (but long and boring, I have to say) and from a pure computer scientist view it's great in my opinion, abstraction but pretty much following the literature in the field, starting of course from the quoted CRLS book we all know and (not) love. Hope this can be useful and do not hesitate to reach me to give me feedback over its content. Also to thank me, it doesn't kill me that much.

2 COURSE PRESENTATION

(Usual general fluff, then the lectures will be present. This is the only slides-based part, found here)

Algorithms have a general motivation: create a network of knowledge and allow, with pacing of times, different development and stories creation, while crafting new solutions. We might define them as a sequence of steps to solve the most mundane problems but also really complex ones.

There are different kinds of *applications*:

- Network routing
- Bioinformatics
- Economics (e.g., game theory)
- Fluid dynamics
- Data mining
- Cryptography
- Machine learning

The point is this: even when making interviews, algorithms are both the logic and the solution to current problems, thinking *repeatedly and abstractly in a concrete (and fast) way*. Historically, there are still a lot of unsolved or still not found problems. That's why the course is *mandatory*.

There are also different *goals*, wanting to introduce advanced principles of algorithm design and analysis. In particular, you'll learn how to:

- Design algorithms for complex domains such as *graphs*
- Recognize "hard" problems and address them using *approximation algorithms*
- Use the power of *randomness* to design fast algorithms
    - and analyze them with appropriate mathematical tools

The *contents* of the course will be the following:

- (Basic) Graph algorithms
    - Graph search and its applications, minimum spanning trees, shortest paths, maximum flows 2 Approximation algorithms
- *Intractable* problems (not solvable in a reasonable amount of time)
    - NP-hardness and reductions between problems
    - Approximation algorithms for intractable problems
        - such as vertex cover, set cover, and the traveling salesperson problem
- Randomized algorithms
    - Main design techniques and analysis tools
        - with applications to problems such as sorting and minimum cuts

Although there are no formal prerequisites, an undergraduate course in algorithms and a good knowledge of (discrete) probability are assumed. Specifically, you should be familiar with:

- *Algorithm design techniques*: divide and conquer, greedy, dynamic programming
- *Data structures*: lists, stacks, queues, binary trees, search trees, heaps
- *Probability*: basic notions, discrete random variables

We want to discuss the *intuition* behind formulating algorithms and distill the core ideas making the algorithms work. Because we are computer scientists, we want to give *rigorousness*: algorithms without proofs are just conjecture and proofs give math logic to those.

We'll follow, in part, an "active learning" approach:

- Will foster and encourage interaction during class
- Will frequently conclude class with 1-2 exercises
    - Whose solution will be shown only at the beginning of next class
- Will frequently post on Moodle further readings
    - News/surveys/research articles/videos related to the topics covered in class
- There is no lab or coding assignments
    - But you are encouraged to code your favorite algorithms up and run them on real data

If you read until here, you sure wanna know: <u>how is the exam?</u>

- Written test, 2 hours. It consists of:
    - 3 questions
        - Theory questions on the topics covered in class
        - Aimed at verifying the student's knowledge of the contents of the course
    - 2 problems
        - Problems whose solution *requires some creativity*
        - Aimed at verifying the student's ability to use concepts
        - Techniques learned during the course to solve new problems

Some more concrete advice: study <u>very well</u> the program of this course; this first graphs part is more of understanding the general rules, focus more on approximation algorithms and Chernoff bounds.

See exams for more and understand yourself the concepts; sometimes he puts concepts he thinks they are easy, or variants of proofs seen during the course in a slightly different way, because he expects you to know <u>exactly</u> the contents of the course, he has no reality grasp on how easy/difficult concepts are. In any case, I told you.

2.1 ACTUAL COURSE PROGRAM

As of 2023/2024 program, these are the lessons. This is made for you in order to be organized with the whole course content:

- Lecture 1 + further reading
    - Course presentation
    - Graphs: the basics

- Lecture 2
    - Graph search and its applications: Depth-First Search, with applications to finding spanning trees, paths, and cycles

- Lecture 3 + further reading
  - More applications of DFS: connectivity and connected components
  - Breadth-First Search, with application to shortest paths

- Lecture 4
  - Minimum spanning trees: a generic algorithm and its correctness
  - Minimum spanning trees: Prim's algorithm

- Lecture 5
  - Minimum spanning trees: Prim's algorithm implemented with heaps
  - Minimum spanning trees: Kruskal's algorithm

- Lecture 6 + further reading
  - The Union-Find data structure
  - Minimum spanning trees: Kruskal's algorithm implemented with Union-Find

- Lecture 7
  - Single-source shortest paths: Dijkstra's algorithm

- Lecture 8 + further reading
  - Single-source shortest paths: the Bellman-Ford algorithm

- Lecture 9
  - All-pairs shortest paths: the Floyd–Warshall algorithm

- Lecture 10 + further reading
  - Maximum flows: the Ford-Fulkerson algorithm

- Lecture 11
  - Complexity classes P and NP
  - NP-hardness and reductions

- Lecture 12 + further reading
  - NP-hardness reductions

- Lecture 13
  - Approximation algorithms
  - Vertex cover: a 2-approximation algorithm

- Lecture 14 + further reading
  - The traveling salesperson problem (TSP): inapproximability & special case metric TSP

- Lecture 15
  - Metric TSP: a 2-approximation algorithm

- Lecture 16 + further reading
    - Metric TSP: a 3/2-approximation algorithm

- Lecture 17
    - Set cover: an O(log n)-approximation algorithm

- Lecture 18
    - Randomized algorithms: motivation and basic notions

- Lecture 19
    - Markov inequalities and their implications
    - Minimum cuts: Karger's algorithm

- Lecture 20
    - Analysis of Karger's algorithm

- Lecture 21
    - Chernoff bounds
    - Applications of Chernoff bounds: coin flips

- Lecture 22
    - Applications of Chernoff bounds: analysis in high probability of Randomized Quicksort

- Lecture 23
    - Applications of Chernoff bounds: exit polls and load balancing

- Lecture 24
    - Exercises

3 1ST PART OF THE COURSE - GRAPHS

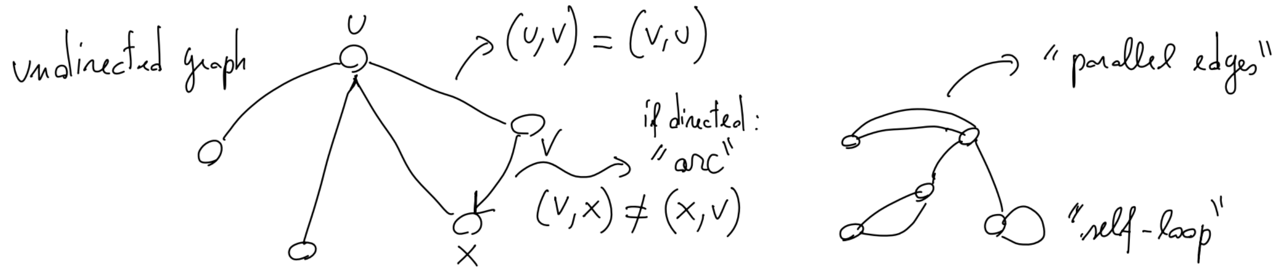
(Suggested readings: The Algorithm, idiom of modern science [here])

A graph is a repartition of the relationships between pairs of objects. In particular, we note:

- $G = (V, E)$ as the graph itself
  - V = set of vertices (a.k.a nodes)
  - $E \subseteq V \; x \; V$ (cartesian product = all) is a collection of edges
    - An edge is a pair of vertices (u, v)
      - It indicates the connection between two nodes
      - A connection of vertices allows for repetition

In the following drawings, we find:

- Directed graphs, which happens if $(u, v) \neq (v, u)$
- Undirected graphs, which happens if $(u, v) = (v, u)$
- Arc = edge inside directed graphs (also called *directed edges*)



In this case, we'll (mostly) use simple graphs, meaning:

- No parallel edges
- No self-loops

3.1 TERMINOLOGY AND CONCEPTS

We give some *terminology*:

- Given an edge $e = (u, v)$
  - e is incident on u and v (happens if vertex if one of endpoints in that edge)
  - u and v are adjacent (there is an edge between the two vertices)
- Neighbors of a vertex v: all vertices v s.t. $(u, v) \in E$
  - all vertices directly connected to a given vertex by an edge
- Degree of a vertex v, denoted as $d(v)$ or $degree(v)$
  - the number of edges incident on v

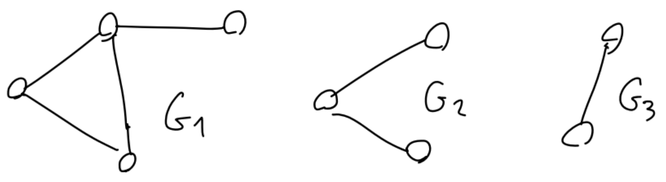
In many ways, graphs are the main modality of data we receive from nature and here we give some *examples* from nature:

- Road networks → (cities, roads)
- Computer networks → (computers, connections)

- World Wide Web (WWW) → (webpages, hyperlinks)
- Social networks (people, friendship relationships)
- Biological networks
    - e.g., molecules (atoms, chemical bonds)
    - e.g., brain (neurons, synapses)
- Finance → (accounts, transactions)

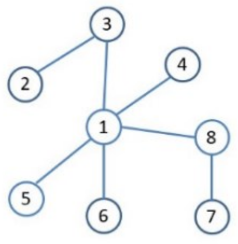
We give some concepts also:

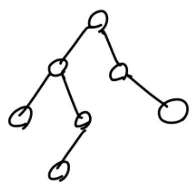
- Path: $u_1, u_2 \dots u_k$ and $(u_i, u_{i+1}) \in E, \forall\, 1 \le i \le k$
    - finite/infinite sequence of nodes which joins a sequence of vertices via edges
- Simple path: u_i (all vertices) are all distinct
    - same definition as above and vertices/nodes are all distinct/so are the edges
    - e.g., 5,1,8,7,6,1,4 has 1 repeated twice so it's not simple
- Cycle: simple path s.t. $u_1 = u_k$ (starts from a given vertex/ends at same node)
- Subgraph: $G' = (V', E')\ s.t.$
    - $V' \subseteq V$
    - $E' \subseteq E$
    - the edges of E' are incident only on vertices of V'
    - in words: it is a subset of the larger original graph
- Spanning subgraph: a subgraph with $V' = V$
    - a subgraph which "spans" the original graph (so there are all the vertices)
    - following other definitions
        - subgraph obtained by edge deletions only but retaining all vertices
        - so it's a subgraph of G with same vertex set as G
- Connected graph: if $\forall u, v \in V, \exists$ a path from u to v
- Connected components: a partition of G in subgraphs $G_i = (V_i, E_i), \forall\, 1 \le i \le k\ s.t.$
    - G_i is connected $\forall i$
    - $V = V_1 \cup V_2 \cup \dots \cup V_k$
    - $E = E_1 \cup E_2 \cup \dots \cup E_k$
    - $\forall i \ne j$ there is no edge between V_i and V_j



G connected $\Rightarrow K=1$

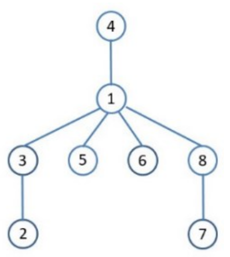
- Tree: connected graph without cycles
    - any two vertices are connected by *exactly* one path





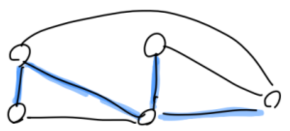
There is also the concept of *rooted tree*:

- There is a root $r \in V$
- There is a father for each non-root node and each node is directly linked to the father
  - $\forall u \in V, u \neq r, \exists!\, p(u)$
- Going father to father, we reach r

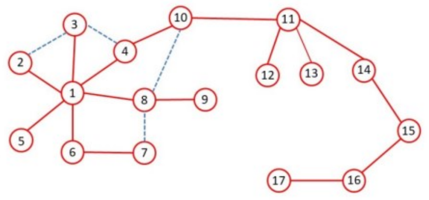

Rooted Tree

Continuing with definitions:

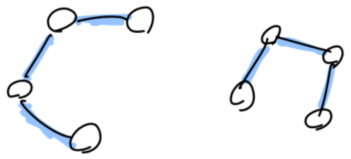
- Forest: set of trees (disjoint)
  - also = undirected graph in which any two vertices are connected by *at most* one path
- Spanning tree: a connected and acyclic spanning subgraph

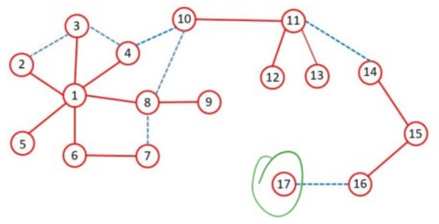


(it exists only if G is connected)



- Spanning forest: a spanning subgraph without cycles





3.2 Basic Problems, Notations and Properties

There are different *basic problems*:

- *Traversal* (systematic exploring of graph e.g., crawling)
- *Connectivity* (tell if the graph is connected or not e.g., wireless networks)
- *Computing connected components* (e.g., wireless networks)
- *(Minimum) spanning trees* (e.g., efficient broadcasting in wireless networks)
- *Minimum-weight spanning trees* (e.g., navigator)
- *Shortest paths* (e.g., social media friend analysis or navigation systems)

Also consider some notations and properties:

- $n = |V|$ (number of nodes)
- $m = |E|$ (number of edges)
  - *Note*: the book uses only $|V|$ and $|E|$
- Size of a graph is $n + m$
  - m is not enough (normally online you would find the size it's $|E|$ = count of edges)
  - Consider a scenario of a graph with n vertices and no edges
  - The size of graph would be n but we don't consider vertices
    - we are accounting for both the "space" and the "connections" occupied

3.2.1 Exercises

Exercise (Properties of graphs)

Let $G = (V, E)$ be a simple, connected graph with n vertices and m edges. Then:

1) $\sum_{v \in V} d(v) = 2m$
2) $m \leq \binom{n}{2}$
3) *G is a tree* $\Rightarrow m = n - 1$
4) *G is connected* $\Rightarrow m \geq n - 1$
5) *G is acyclic (i.e., is a forest)* $\Rightarrow m \leq n - 1$

Prove the previous properties.

Solution (*Note*: the professor is lapidary in his solutions, which I hate – so, apart from his official solution, past years explanations plus general theory is incorporated to give more precision to proofs)

1) In the summation, every edge is counted exactly twice
   a. This is a famous result, called "handshaking lemma"

2) In a simple graph, there are $\binom{n}{2}$ possible pairs of vertices. Specifically, $m \leq \binom{n}{2} = \frac{n(n-1)}{2}$

3) Fix a root on a vertex (so, consider G as rooted tree, thanks to the equivalence between rooted tree and "free" tree). Then E represent father-child relationships, which are $n - 1$ (which means each non-root node has a unique father), because one parent for each $u \neq 2$ root (given $E = \{(u, p(u)) \mid u \neq r\}$) has no parent $|E| = n - 1$

4) G is a tree that may have cycles $\Rightarrow$ it can only have more edges than a tree
   a. Consider connectivity removes edges and keeps the graph connected without cycles, thanks to $n - 1$ edges

   b. Alternatively, consider the following loop:

```
while ∃ $loop\ C$: do
        $remove\ one\ edge\ of\ C\ from\ G$
end while
```

   So, here: G is connected after each iteration and at the end of the while, G is acyclic and connected $\rightarrow$ G is a tree $\rightarrow$ $m' = n - 1 \leq n$

5) G is a tree that may not be connected $\Rightarrow$ it can only have less edges than a tree
   a. If it is a tree without cycles, it is a forest, and its maximum edges are $n - 1$

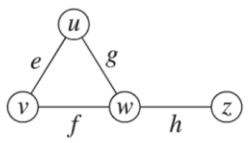
Completely:

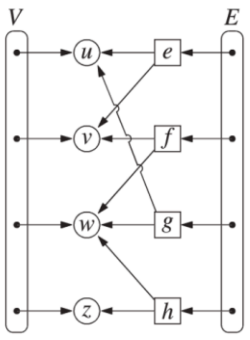
G is a made of $k \geq 1$ trees, the i^{th} tree has n_i nodes and m_i edges, with $n = \sum_{i=1}^{k} n_i$, $m_i = n_1 - 1$ and $m = \sum_{i=1}^{k} m_i = \sum_{i=1}^{k} (u_i - 1) = n - k \leq n - 1$.

3.3 Representing a graph

How to encode a graph for use in an algorithm?

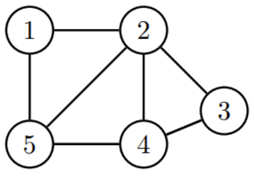
Consider a list of vertices L_V and a list of edges L_E. (they contain all information about E and V and the links between each other). Let's consider vertices are called $1, 2, \dots n$. This is useful but does not allow for fast algorithms overall. Here we have examples of both pointers:





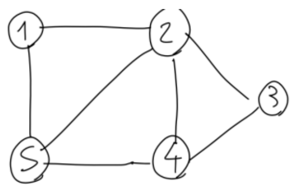
To allow for *direct access to edges*, one of the following data structures are used, in addition to pointers to L_V, L_E:

- Adjacency list
  - An array A of n lists, one ∀ vertex $v \in V$ (consider the example below)
  - Each containing all the vertices adjacent to v (represented by table below)



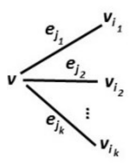
1	2,5
2	1,3,4,5
3	2,4
4	2,5,3
5	4,1,2

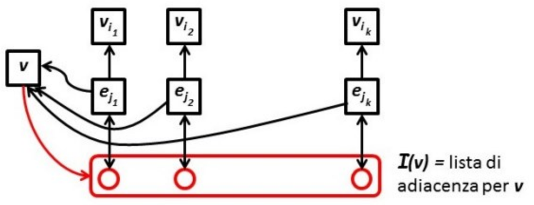
example:



A =

1	2 5
2	1 3 4 5
3	2 4
4	2 5 3
5	4 1 2



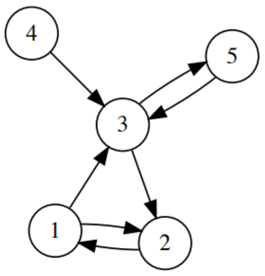


What if directed? Only vertices pointed for that vertex.

- *Pro*:
  - Space usage $\theta(n + m)$ i.e. linear in the size of the graph
- *Con*:
  - No quick way to determine if a given edge is in the graph

- <u>Adjacency matrix</u>
    - A $n \times n$ matrix A s.t. $A[i,j] = \begin{cases} 1, & if\ edge\ (i,j) \in E \\ 0, & otherwise \end{cases}$

$$A = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array} \longrightarrow \text{symmetric}$$



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 1 & 0 & 0 \end{array}$$

- If graph is directed → the matrix is *asymmetric*
- If graph is undirected → the matrix is *symmetric*
    - Edges are bidirectional → only half of matrix needs to be stored
    - Operations for this kind of matrix are more efficient in general

In case of a *weighted graph*, each cell of the matrix has either the value of the edge weight (as number) w or $-/null$ to represent null costs. This kind of graph represents costs, capacities, etc.

- *Pro*:
    - Quick to determine if a given edge is present
- *Con*:
    - Space required is $\theta(n^2)$ → can be superlinear in the input size
    - if number of vertices increases, the space required by matrix grows quadratically

It may also depend on the number of edges:

- *Dense* graph = number of edges close to maximal number → $m = O(n^2)$
    - many cells inside adjacency matrix will be populated by non-zero values
    - adjacency matrix is mostly used here
        - allows to quickly test the presence of an edge and check its info
- *Sparse* graph = number of edges with only a few edges → $m = O(n)$
    - conversely, majority of values will be zero

3.4 Graphs Algorithms

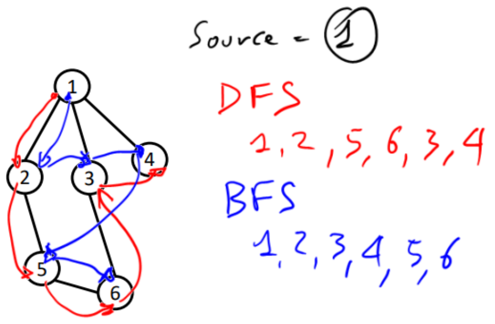
We are focusing over *graph search and its applications*, in particular traversal/exploration. They provide a systematic way to explore a graph starting from a vertex $s \in V$ ($s =$ source vertex) visiting all the vertices (starting from a graph and a source vertex) – basically design patterns to solve specific problems.

Even using only the lists L_V and L_E, we explore the full graph and given this is not systematic, the exploring is not exploitable to solve problems.

The most famous algorithms are:

- *Depth-First Search (DFS)* → aggressive, goes in depth, then comes back and so forth
- *Breadth-First Search (BFS)* → non-aggressive, proceeding by levels inside graph

In particular, consider the following graphs; in each, the types of visits are defined already in color.



Observe that:

- DFS and BFS serve as design patterns, acting as building blocks
    - where the visit operation can be instantiated to solve specific problems
    - such as connectivity and spanning tree identification
- Traversing L_V and L_E lists also achieves complete graph exploration
    - however, the lack of systematic exploration makes it less useful for problem-solving
- The idea behind both is to prioritize visiting neighbors with lower IDs from the starting vertex

3.4.1 Depth-First Search (DFS)

(Further readings for this one: paper and survey)

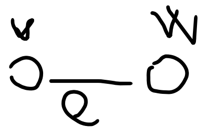
This is a recursive algorithm which:

- Starting from a source $s \in V$ "visits" all vertices of the connected component $C_S \subseteq G$ containing s
- Uses adjacency list as graph representation
- Every vertex v has a field $L_v[v].ID$ which can be either
    - 1 if visited
    - 0 otherwise
- Every edge e has a label $L_E[e].Label$ which can be either
    - $null$ initially

    - $DISCOVERY\ EDGE$ or $BACK\ EDGE$
        - first label indicates an edge which allows discovery of vertices
        - second label indicates non-tree edges
            - that go from a node u in the DFS tree
            - to some ancestor w of u in the DFS tree
            - this kind of edges is useful in order to find cycles
- In all moments when executing the algorithm, vertex u is *discoverable* from a vertex v if there exists a path from v to u made of not-yet-visited vertices

Consider the following procedure (works for both directed and undirected graphs):

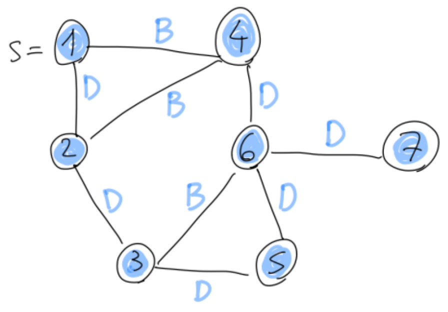
```
procedure $DFS(G,V)$                    (first invoke: $v = s$ → source vertex)
    $visit\ v$
    $L_V[v].ID = 1$
    for all $e \in G.incidentEdges(v)$: do
        if $L_E(e).label = null$ then
            $w = G.opposite(v,e)$
            if $L_V[w].ID = 0$ then
                $L_E[e].label = DISCOVERY\ EDGE$
                $DFS(G,w)$
        else
            $L_E[e].label = BACK\ EDGE$
```

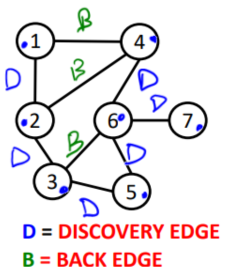


Because I like people understanding stuff, let's comment human-like this code, considering we:

- Take each vertex and we see if it was visited or not
    - this is done *on the connected component* touching all vertices and edges
    - we use adjacency lists to induce an order of visit in neighbors
- Check if the current vertex (with ID field) has been visited or not
- Loop on all edges incident to current vertex – iterator of the neighbors of v
- Check if label of current edge was not labeled = it was not explored
- Consider the opposite vertex = other endpoint of the edge
- If that opposite vertex has not been visited yet
    - edge leads to an unexplored vertex, indicating a discovery edge
        - a vertex is discoverable if there exists a path between u and v not visited
    - this will be labeled, indicating it's the first time the edge is being traversed
    - then we recursively call the algorithm to explore the connected component
- Else (aka it was already visited)
    - the edge is leading to an already explored vertex
    - the edge is labeled indicating a connection back to the ancestor

The following is an example of the algorithm being applied:



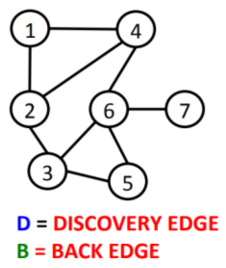


s = 1

Adjacency lists (increasing order)

1:	2	4		
2:	1	3	4	
3:	2	5	6	
4:	1	2	6	
5:	3	6		
6:	3	4	5	7
7:	6			

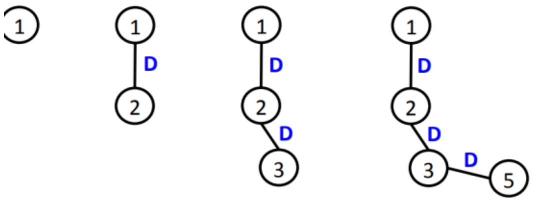
To be complete:

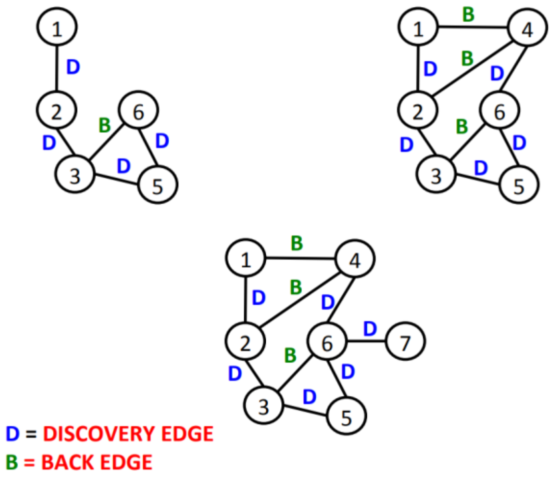


s = 1

Liste di adiacenza (ordine crescente)

1:	2	4		
2:	1	3	4	
3:	2	5	6	
4:	1	2	6	
5:	3	6		
6:	3	4	5	7
7:	6			





3.4.1.1 *Correctness*

At the end of the algorithm:

1) all the vertices of C_s have been visited and all the edges in C_s are labelled either $DISCOVERY/BACK\ EDGES$
2) the set of $DISCOVERY\ EDGES$ is a spanning tree T of C_s called "DFS tree"

Proof:

1) (short: by construction)
   By contradiction, $\exists v \in C_s$ not visited. Since C_s is connected, there is a path from s to u

   $s = u_0 \to u_1 \to u_2, \dots, u_k = u$. Let u_i be the first unvisited vertex in the path ($u_j \in C_s\ \forall j$)

   We run into the contradiction: $DFS(G, u_{i-1})$ must have been executed and therefore $DFS(G, u_i)$ is called (meaning u_i was found not visited). This happens in contradiction to the hypothesis (it's not possible to find a vertex unvisited and marked as such).

   A vertex u is visited only when $DFS(G, u)$ is invoked ⇒ DFS is called $\forall v \in C_s$ ⇒ all incident edges on v are labelled, by construction.

2) DFS is called $\forall v \in C_s$, once, and $\forall v \neq s$, $\exists$ a vertex u s.t. (u, v) $\exists$ and is labelled $DISCOVERY\ EDGE$ and $DFS(G, v)$ is invoked from $DFS(G, u)$. We say that v gets "discovered" by u and let's call u "father" of $v \Rightarrow \forall v \in C_s, v \neq s$
    a. $\exists!$ father (there exists a father and it is unique)
    b. going back father to father eventually s is reached

    Then, the set of $DISCOVERY\ EDGES$ is a *rooted tree* that *touches* all the vertices of C_s and it's a spanning tree of C_s (unique path from every vertex to the source one and each is discovered by exactly one parent vertex).

3.4.1.2 Complexity

Given:

- n_s: number of vertices of C_s (one invocation $\forall v \in C_s$)
- m_s: number of edges of C_s (costs related to node, excluding recursive invocations inside)

The complexity overall is:

$$\theta\left(\sum_{v \in C_s} d(v)\right) = \theta(m_s)$$

(sum of degrees of each node proportional to number of edges)

Remark: Note that C_s is connected, so:

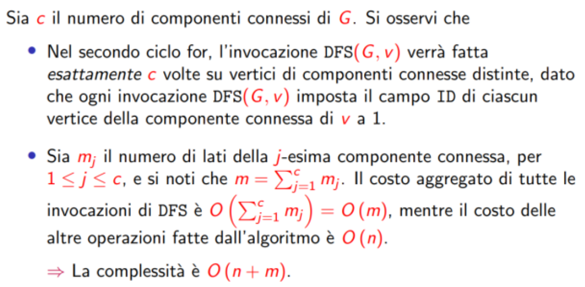
- $m_s \geq n_s - 1$ (connected, so for n vertices we would have at least $n - 1$ edges)
- $m_s = \Omega(n_s)$ (n. of edges at least proportional to n. of vertices)

More in general: $O(n + m)$ – n vertices and m edges

3.4.1.3 Extension

The possible extension is to visit <u>all</u> the graph (aka: all components even if not connected) – following pseudocode can be used as design pattern to extend to all graph in case it's not connected:

```
for $v = 1\ to\ n$ do:
    $L_v[v].ID = 0$
for $v = 1\ to\ n$ do:
    if $L_V[v].ID = 0$ then
        $DFS(G, v)$
```



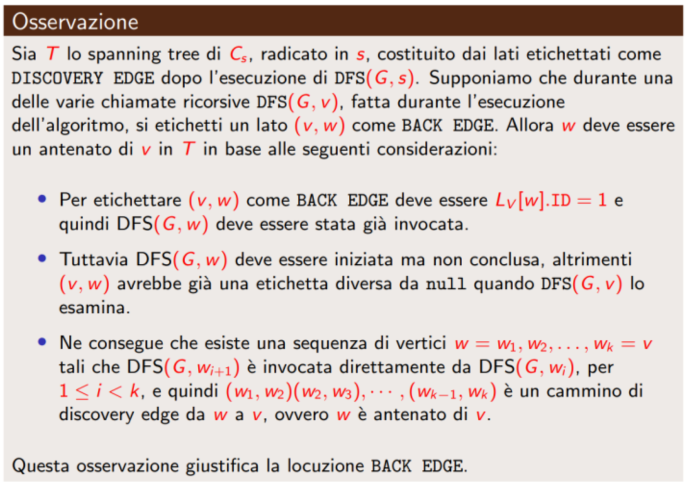
Sia c il numero di componenti connessi di G. Si osservi che

- Nel secondo ciclo for, l'invocazione DFS(G, v) verrà fatta *esattamente* c volte su vertici di componenti connesse distinte, dato che ogni invocazione DFS(G, v) imposta il campo ID di ciascun vertice della componente connessa di v a 1.
- Sia m_j il numero di lati della j-esima componente connessa, per $1 \leq j \leq c$, e si noti che $m = \sum_{j=1}^{c} m_j$. Il costo aggregato di tutte le invocazioni di DFS è $O\left(\sum_{j=1}^{c} m_j\right) = O(m)$, mentre il costo delle altre operazioni fatte dall'algoritmo è $O(n)$.

  $\Rightarrow$ La complessità è $O(n + m)$.

Overall, the complexity if $\theta(n + m)$ because it scans over all the vertices and nodes.

Note: BACK EDGE is justified by the following – full proof before and examples below kept in Italian since was in old notes and not useful here, but to be complete.

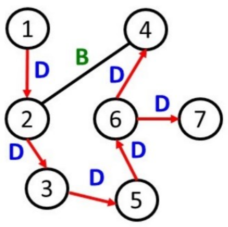


Osservazione

Sia T lo spanning tree di C_s, radicato in s, costituito dai lati etichettati come DISCOVERY EDGE dopo l'esecuzione di DFS(G, s). Supponiamo che durante una delle varie chiamate ricorsive DFS(G, v), fatta durante l'esecuzione dell'algoritmo, si etichetti un lato (v, w) come BACK EDGE. Allora w deve essere un antenato di v in T in base alle seguenti considerazioni:

- Per etichettare (v, w) come BACK EDGE deve essere $L_V[w].ID = 1$ e quindi DFS(G, w) deve essere stata già invocata.
- Tuttavia DFS(G, w) deve essere iniziata ma non conclusa, altrimenti (v, w) avrebbe già una etichetta diversa da null quando DFS(G, v) lo esamina.
- Ne consegue che esiste una sequenza di vertici $w = w_1, w_2, \dots, w_k = v$ tali che DFS(G, w_{i+1}) è invocata direttamente da DFS(G, w_i), per $1 \leq i < k$, e quindi $(w_1, w_2)(w_2, w_3), \cdots, (w_{k-1}, w_k)$ è un cammino di discovery edge da w a v, ovvero w è antenato di v.

Questa osservazione giustifica la locuzione BACK EDGE.



Spanning tree dei discovery edge:
- il verso delle frecce rappresenta le relazioni padre-figlio
- Es.: (4,2) è marcato come BACK EDGE e 2 è antenato di 4

3.4.1.4 Exercises

1) *Given a graph G and two vertices s, t determine, if it exists, a path from s to v*
2) *Given a graph G return a cycle (if any)*

Solution

- 1st exercise ($s - t$ path/connectivity)
  - $\forall v \in V$ add a field $L_V[v].parent$
  - Modify $DFS(G, v)$ s.t. when a $DISCOVERY\ EDGE$ (v, w) is labeled
    - then $L_V[w].parent = v$ (v is parent of w in $DISCOVERY\ EDGE$ tree)
  - Run $DFS(G, s)$. Check if t has been visited
    - NO: then return "No path"
    - YES: starting from t, follow the "parent" label, so as to build a path from t to s
  - Complexity: $O(m_s)$ where m_s is the number of edges of s connected component

- 2nd exercise (cycle) → we go back thanks to back edges because they "close" the cycles
  - $\forall v \in V$ add a field $L_V[v].parent$ and $\forall e \in E$ add a field $L_E[e].ancestor$
  - (v, w) is a $DISCOVERY\ EDGE$ then $L_V[w].parent = v$
  - (v, w) is a $BACK\ EDGE$ then $L_E[e].ancestor = w$
    - then w is an ancestor of v in the DFS tree
  - Run DFS on each connected component
  - Check all the edges
    - as soon as an edge $e = (v, w)$ is found as $BACK\ EDGE$
    - and $L_E[e].ancestor = w$
    - then return a cycle adding to e all the edges found in the path from v to w
    - if no $BACK\ EDGE$ is found, then return "No Cycles" (it would be a tree)

Complexity for both algorithms: $\theta(n + m)$ → invoked DFS once for each connected component

3.4.1.5 More applications

More or less what we did until now with DFS was returning a spanning tree. Other problems which can be solved are the following:

- *Graph connectivity*: return whether the graph is connected or not
- *Connected components*: return a labeling of all the vertices of G s.t. 2 vertices have the same label if and only if they are in the same connected component – basically, keeping a count of connected components – if it is 1, then G is connected

We will modify the algorithm in such a way that ~~$L_V[v].ID = 1$~~ (let's generalize it, so) $L_V[v].ID = k$ (integer, label of the k-th component), counting the connected component of G and assigning the same ID to vertices in the same connected component.

```
for $v = 1\ to\ n$ do:
    $L_V[v].ID = 0$
$k = 0$
for $v = 1\ to\ n$ do:                 Connected          Graph
                                       components         connectivity
    if $L_V[v].ID = 0$ then
        $k = k + 1$
        $DFS(G, v, k)$
if $k = 1$ then return $YES$
return $NO$
```

Complexity of the whole thing: $\theta(n + m)$

To be complete (*analysis*):

Let c be the number of connected components in G. Then:

- The call $DFS(G, v, k)$ is executed exactly c times on vertices from different connected components. Then, at the end of the for loop $k = c$ ($k = 1$ if G is connected)
  - ⇒ The algorithm is correct
- We can prove that the complexity is $O\ (n\ +\ m)$ as before.

3.4.1.6 Summary

Given a graph G, the following problems can be solved in $\theta(n + m)$ using DFS:

- Test if G is connected
- Find the connected components of G
- Find a spanning tree of G (if G is connected – otherwise it's called spanning forest)
  - execute $DFS(G, s)$ from any vertex s and return $DISCOVERY\ EDGES$ as spanning tree edges and given DFS is executed once, its complexity is $O(m)$
- Find a path between two vertices (if any) – so from a vertex s to t (s-t connectivity)
- Find a cycle (if any)

Written by Gabriel R.

Possible questions for exam: Show how to find a spanning tree in linear time/Find if graph is connected in linear time, something like that.

3.4.2 Breadth-First Search (BFS)

This is an iterative algorithm that starting from a source vertex "visits" all the vertices in the same connected component of s, and partitioning the vertices in levels L_i depending on their distance i from s (with distance we mean the shortest path). We'll use adjacency list to represent G:

- $L_V[v].ID = 0$ if not visited, 1 if visited
- $L_E[e].label = null$ if e has no label, $DISCOVERY/CROSS\ EDGE$
    - $CROSS\ EDGES$ connect vertices at different levels (different labels)

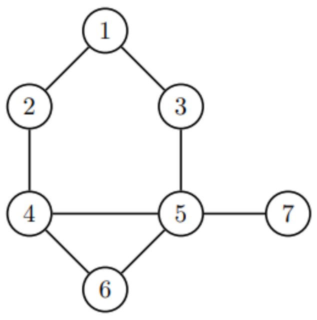
```
procedure $BFS(G, s)$
    $visit(s)$
    $L_V[s].ID = 1$
    $Create\ a\ set\ L_0\ containing\ s$
    $i = 0$
    while $(!L_i.isEmpty())$ do:
        $Create\ a\ empty\ set\ of vertices\ L_{i+1}$
        for each $v \in L_i$ do:
            for each $e \in G.incidentEdges(v)$ do:
            if $L_E[e].label = null$ then
                $w = G.opposite(v, e)$
                if $L_v[w].ID = 0$ then
                    $L_E[e].label = DISCOVERY\ EDGE$
                    $visit\ w$
                    $L_V[w].ID = 1$
                    $add\ w\ in\ L_{i+1}$
                else
                    $L_E[e].label = CROSS\ EDGE$
    $i = i + 1$
```

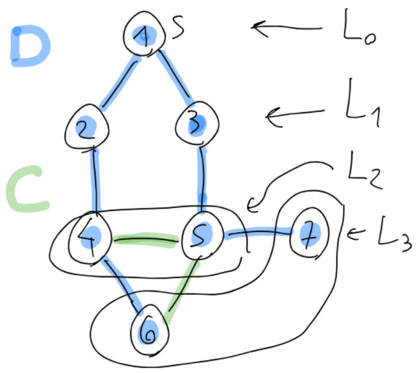
An explanation step-by-step of the algorithm:

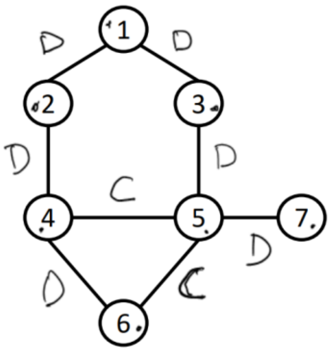
- Visit the source vertex
- Iterate over all levels
- Explore neighbors starting from level L_i
- Create a set of vertices for the next level L_{i+1}

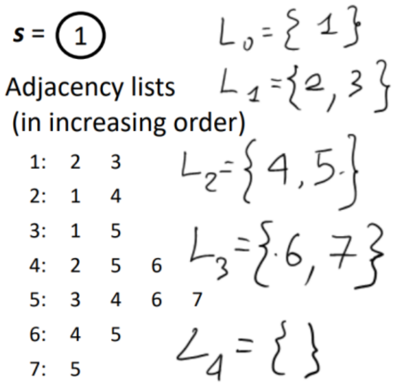
- For all incident edges
    - if vertex has not been visited
    - we get the opposite vertex
    - If vertex has not been visited yet
        - mark edge as $DISCOVERY\ EDGE$
        - visit vertex
        - mark it as visited
        - add the vertex to the set of vertices for the next level
    - Else (if vertex has already been visited)
        - node represents crossing between different levels
        - it will be marked as $CROSS\ EDGE$
        - it connects two nodes that don't share any ancestor-descendant relation
    - Increment the level counter
    - Algorithm terminates where there are no more vertices to visit

Here we do the following example (at first invoke: $BFS(G, 1)$):







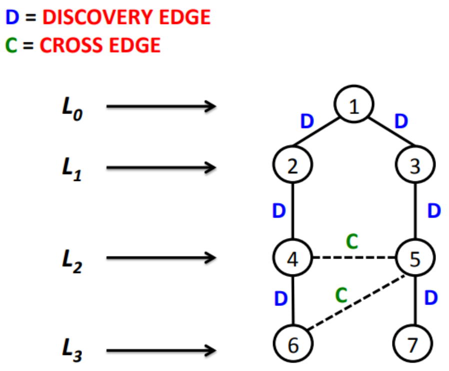


$s = 1$

Adjacency lists
(in increasing order)

1:	2	3		
2:	1	4		
3:	1	5		
4:	2	5	6	
5:	3	4	6	7
6:	4	5		
7:	5			

$L_0 = \{1\}$
$L_1 = \{2, 3\}$
$L_2 = \{4, 5\}$
$L_3 = \{6, 7\}$
$L_4 = \{\}$



The algorithm, when executed, it will behave like the following:

- $L_0 = \{1\}$
    - takes source vertex and marks it as visited
- $L_1 = \{2, 3\}$
    - goes to next level and for each vertex takes its connected component (incident edge)
    - so, from 1 goes to 2 because it's connected
        - given it was not explored yet, also jumps to 3
- $L_2 = \{4, 5\}$
    - same thing for 4 and 5
    - they were the connected components of 2 and 3, jumping to the opposite edge

- $L_3 = \{6, 7\}$
    - from 5 we see a connected component, found within next level of adjacency list
    - given they were already explored, between 4,5/5,6 two $CROSS\ EDGES$ are discovered

3.4.2.1 Correctness

At the end of $BFS(G, s)$ we have:

1) all vertices in C_s are visited and all edges are labelled $DISCOVERY/CROSS\ EDGE$
2) the set of $DISCOVERY\ EDGES$ are a spanning tree T (tree touching all the vertices) of C_s
    a. analogously to DFS, we call it BFS tree (this is rooted in s)
3) $\forall v \in L_i$ the path in T from s to v has i edges and every other path from s to v has exactly i edges (e.g., $\geq i$ edges) $\approx i \equiv distance(s, v)$

Proof of (1) and (2): same as for the DFS

Proof of (3):

- Let $P: s = u_0 \rightarrow u_1 \rightarrow \cdots \rightarrow u_i = v$ where $u_j \in L_j$ is "discovered" from U_{j-1} $\forall j \Rightarrow (u_{j-1}, u_j)$ is a $DISCOVERY\ EDGE \Rightarrow P$ is a path of T
- By contradiction, assume $\exists$ a path $P': s = z_0 \rightarrow z_1 \rightarrow \cdots z_t = v$ with $t < i$ (shorter)
- This implies that $s = z_0 \in L_0, z_1 \in L_1, z_2 \in L_1\ or\ L_2 \ldots z_t \in L_1$ or L_2 or … L_t (might be on some levels before)
- This means that, since $z_t = v, v \notin L_i$ but this is a contradiction

In words:

- What you have is that the first node is in level zero, so on and so forth until the tth node is in some list between L_1 and L_t
- If this were true, then we have that $v \neq L_i$, because we have that $t < i$
- This is absurd since we assumed that $v \in L_i$

3.4.2.2 Complexity and applications

$\forall v \in C_s$ there is one iteration of the first `for` loop and $d(v)$ (or $degree(v)$) iterations of the second `for` loop (inside of it and for all access to L_i, only $\theta(1)$ complexity).

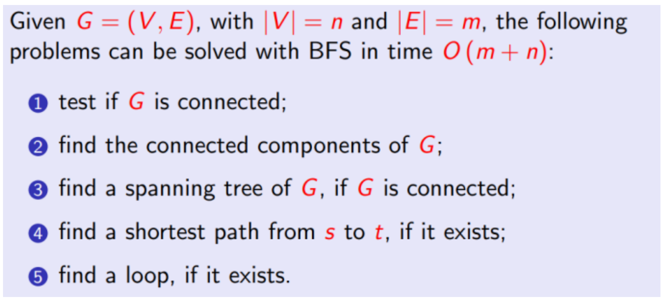
Complexity:

- $\theta(m_s)$ (which becomes $\theta(m)$ if G is connected; in general, each for execution and access to lists)
- again, more in general $O(n + m)$

Some applications:

1) Same as for DFS in $\theta(n + m)$ time

Specifically, this means the following:



Given $G = (V, E)$, with $|V| = n$ and $|E| = m$, the following problems can be solved with BFS in time $O(m + n)$:

1. test if G is connected;
2. find the connected components of G;
3. find a spanning tree of G, if G is connected;
4. find a shortest path from s to t, if it exists;
5. find a loop, if it exists.

2) Given a graph $G = (V, E)$ and $s, t \in V$ return the *shortest* path (in terms of least amount of edges between two nodes) from s to v (if any)
   a. $\forall v \in V, L_V[v].parent$
   b. modify $BFS(G, s)$ s.t. when (v, u) is labeled $DISCOVERY\ EDGE$ then $L_V[v].parent = v$
   c. run BFS and return the set of child-parent edges
   d. Complexity: $\theta(m_s)$

More precisely:

- Visits the full graph G (even if it is not connected)
- Determine the number of connected components in G
- If G is connected, find a spanning tree of G

3.5 Problems solvable with DFS and BFS

Given in an exam was asked, here we summarize this in a specific section:

- test if graph is connected
- find connected components
- find a $s - t$ path
- find a cycle if it exists
- find a spanning tree, if graph is connected

Complexity for both: $O(n + m)$

4 Minimum Spanning Tree (MST)

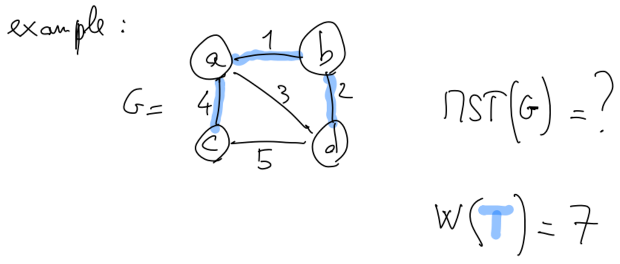
The goal is to interconnect a set of objects in the cheapest possible way (e.g., connecting PCs inside departments using the least amount of cable as possible). It's a fundamental between the computation problems, studied since the 1920s. (See for reference §Chapter 21 of Fourth Edition of CRLS)

4.1 Definition and Applications

More specifically, its *definition* is the following:

- *Input*: A graph $G = (V, E)$ undirected, connected and weighted
    - A weight function $w: E \to \mathbb{R}$ where $w(u, v) =$ cost of edge (u, v)
    - The bigger the weight, the bigger to cost to pay when traversing that edge
- *Output*: A spanning tree $T \subseteq E$ of G s.t. $w(t) = \sum_{u,v \in T} w(u, v)$ is minimized
    - Goal is minimizing the sum for all weights of every edge of the tree

Consider the following example; here, the MST is made of the blue part (minimum part to cover all vertices – for the sake of simplicity, we consider it starting from a):



We give the following observations:

- Minimum-~~weight~~ spanning tree (means minimum ST → minimum-weight ST)
    - This is not to be confused with e.g. minimum number of sides
    - Because *all* spanning trees have the same number of sides
        - and we want the one that weighs the least (it may not be unique)
- Connected assumption is without loss of generality (wlog)
    - If graph is not connected, we talk about Minimum Spanning Forest (MSF)
        - = a MST for each connected component

There are different *applications* we can define:

- Networks (computers, sensors, electrical)
    - E.g., broadcast determining a backbone
        - a subgraph connecting all network nodes and with minimum cost
- Machine learning (building block for clustering algorithms)
- Computer vision (object detection)
- Data mining
- Subroutine in other (approximation) algorithms
    - To solve other problems, e.g. TSP

4.1.1 Difficult in determining an MST

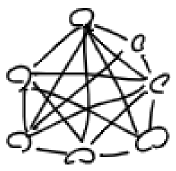
We ask ourselves some *questions*:

- How difficult is it?
    - The number of possible solutions can be exponential w.r.t the input size
- How many spanning trees can a graph have?

The simplest MST algorithm is to enumerate all the spanning trees (STs) and select the one with minimum weight. We would need a *complete graph* to do that: it has all $\binom{n}{2}$ possible edges.

A *complete* graph has $n^{(n-2)}$ different STs (worst case would be exponential; when $n \geq 50$ quantity larger than the number of atoms in the known universe, even on the small graphs this would not work).

The right figure shows a complete graph.



However, surprisingly, MST can be solved in near-linear time (specifically in $O((n+m)\ log\ n)$! It can be done using greedy algorithms ⇒ simpler to understand and implement in practice (e.g., Prim, Kruskal). They both apply (in different ways) a *generic greedy algorithm*, but with different applications.

4.2 Generic Greedy Algorithm for MST

The idea of a generic-MST algorithm is to maintain the following invariant:

- At each iteration, A is a subset of edges of some MST
- Every time an edge is chosen, this is considered to be a right edge
    - Because the choices are made in a greedy manner
- At each iteration, the algorithm adds an edge that does not violate the invariant
    - considered "safe" edge for A (safe to add it/don't do a mistake if you do)

```
procedure $Generic - MST(G)$
    $A = \emptyset$
    while $A\ does\ not\ form\ a\ spanning\ tree$ do:
        $find\ an\ edge\ (u, v)\ that\ is\ safe\ for\ A$   $//\ crucial\ step$
        $A = A \cup \{(u, v)\}$          $//\ add\ vertices\ to\ A$
    return $A$          $//\ A\ is\ an\ MST$
```

This is simple but does not say anything on how to find a "safe" edge. So, the question is exactly:

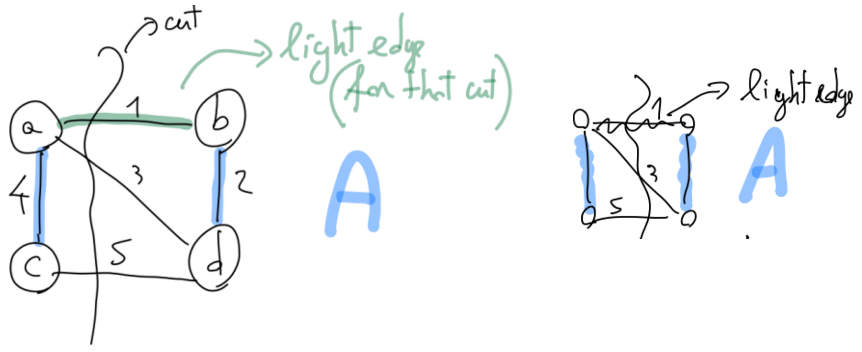
- How to find a safe edge?

Luckily, MSTs enjoy the structural property given by the Theorem next. First, some *definitions*:

- A <u>cut</u> of graph $G = (V, E)$ is a partition of $V \to (S, V \setminus S)$ (figure on the right)
    - in words, a partition of vertices into two disjoint subsets
    - it can be done on one or more edges
- An edge $(u, v) \in E$ <u>crosses</u> a cut $(S, V \setminus S)$ if $u \in S$ and $v \in V \setminus S$ (or viceversa, so $v \in S$ and $u \in V \setminus S$)
    - so, if its endpoints lie in different subsets of the partition defined by the cut
- A cut <u>respects</u> a set of edges A if no edge of A *crosses* the cut
- Given a cut, *an* edge that crosses the cut and is of minimum weight is called <u>light edge</u> (for that cut) → they are useful, because when included in MSTs, they have minimum weight
    - it's important to say "an" edge given there could be two edges with same weight
    - in some cases, it's called *lightest edge*, at least not in this course

There is also the *minimum cut*, for which we have $d(v) \geq t \; \forall v \in V$, where t is a generic size of graph. Summing up all n vertices, we obtain $\sum_{v \in V} d(v) \geq tn$, concluding it's $\sum_{v \in V} d(v) = 2m$. (this one is an exam question, so I put it here in case it could be useful).

We give the following example; a simple cut on three edges and the light edge is to be considered as such because it's respectful - it doesn't cross the cut and it's the minimum weight:

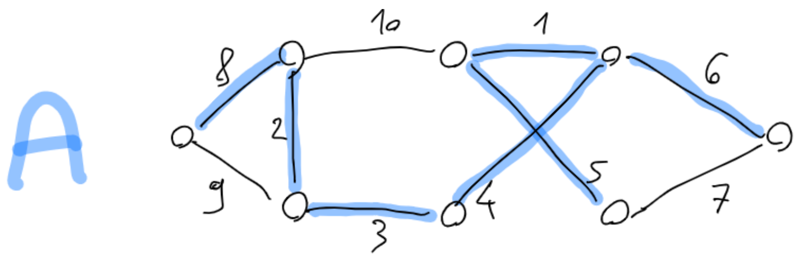


4.2.1 Theorem and Proof

Theorem:

Let $G = (V, E)$ be an undirected, connected and weighted graph. Let A be a subset of E included in some MST of G, let $(S, V \setminus S)$ a cut that respects A and let (u, v) be a light edge for $(S, V \setminus S)$. Then (u, v) is safe for A.

Consider the following example of $Generic - MST$; it basically just throws cuts at random and selects the edge with minimum cost which "respects" the others (aka it was not taken before), connecting possibly all vertices at least once, because it has to "span" them:



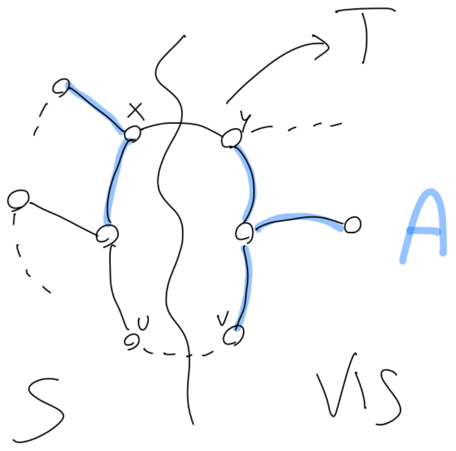
As said, this one considers cuts at random; other algorithms have rules to choose cuts (e.g., Kruskal).

Proof of theorem:

It uses the technique of "cut and paste", standard technique used in the context of greedy algorithms, which is an "exchange argument".

- Cut-and-Paste is a way used in proofing graph theory concepts
    - Idea is this:
        - Assume you have solution for Problem A, you want to say some edge/node, should be available in solution
        - You will assume you have solution without specified edge/node
        - You try to reconstruct a solution by cutting an edge/node
            - then pasting specified edge/node
            - and say new solution benefit is at least as same as previous solution
- *So*: fake to take an optimal solution and transform it in the solution returned by the algorithm
- This shows the cost of the two solutions is the same
    - and also the solution returned by the algorithm it's optimal

Let T be an MST that includes A (basically, we take an optimal solution considering safe edges). Assume that $(u,v) \notin T$ (otherwise, we'd be done). We'll build a new MST T' that includes $A \cup \{(u,v)\}(\Rightarrow (u,v)$ is safe for A). Consider the following example, in which (u,v) is added to T and a cycle is created, because T is a ST.



By hypothesis, (u,v) crosses $(S, V \setminus S) \Rightarrow \exists$ another edge of T that crosses that cut $\Rightarrow (x,y)$ in the figure (it would exist because it's a spanning tree, if there wasn't it wouldn't be in the first place).

- By hypothesis, $(S, V \setminus S)$ respects $A \Rightarrow (x,y) \notin A \Rightarrow$ removing (x,y) from T and adding (u,v) we obtain a new spanning tree $T' = T \setminus \{(x,y)\} \cup \{(u,v)\}$ that includes $A \cup \{(u,v)\}$, as we wanted (because it doesn't partition it and simply adds (u,v) regularly)
- Now we need to show that T' not only is a ST (spanning tree), but also a MST
- (x,y) and (u,v) both cross $(S, V \setminus S)$ but by hypothesis (u,v) is the light edge between the two
    - $\Rightarrow w(u,v) \leq w(x,y) \Rightarrow w(T') = w(T) - \underbrace{w(x,y) + w(u,v)} \leq w(T)$
- But T is an MST $\Rightarrow w(T') = w(T)$

In words: we've shown that adding an edge between vertices to a tree, making it a MST again, maintains its optimality. By proving that the added edge was already a greedy choice, we've shown that its inclusion in the graph does not increase the weight of the tree and it maintains its properties safely.

We'll now see two MST algorithms that organize these "respectful" choices.

Remember one thing overall:

- An MST is unique only for all different weights
- Otherwise, more MSTs exist at a time

4.3 PRIM'S ALGORITHM

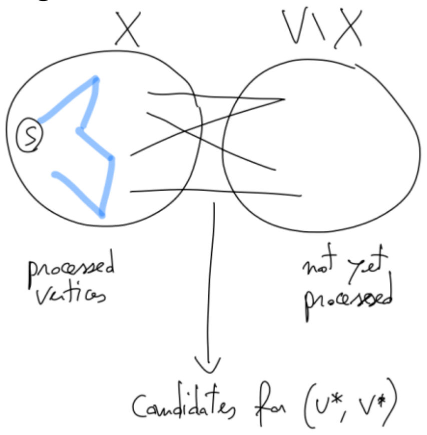
This algorithm was crafted in 1957. How does Prim's algorithm apply $Generic - MST(G)$:

- A is a single tree
- Safe edge: a light edge that connects the tree with a vertex that does <u>not</u> belong to the tree
  - $(S, V \setminus S)$ where S are nodes of A

Here goes the pseudocode:

```
procedure $Prim(G, s)$      $// S = source\ vertex$
    $X = \{s\}$
    $A = \emptyset$
    $// while\ there\ is\ an\ edge\ linking\ X\ with\ other\ things$
    while $there\ is\ an\ edge\ (u, v)\ with\ u \in X\ and\ v \notin X$ do:
        $(u^*, v^*) = a\ minimum\ weight\ such\ edge\ (aka\ light\ edge)$
        $add\ vertex\ v^*\ to\ X\ \ //\ add\ external\ node\ to\ X$
        $add\ edge\ (u^*, v^*)\ to\ A\ \ //\ add\ new\ edge\ just\ found\ to\ A$
    return $A$          $//\ A\ is\ now\ a\ MST$
```

We explain with a figure how this algorithm works:



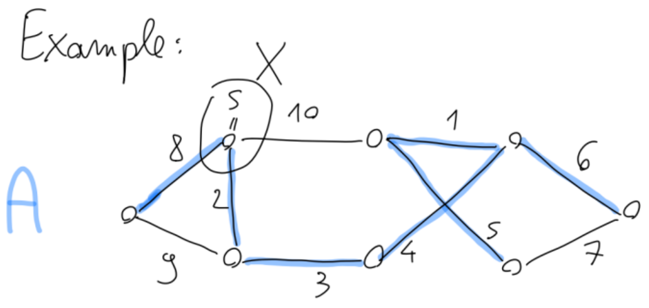
It's also called Jarnik's algorithm, since it was the first to study this problem, then discovered independently by both Dijkstra and Prim (so you would find inside "Algorithms" by Jeff Erickson – next page an example with this name).

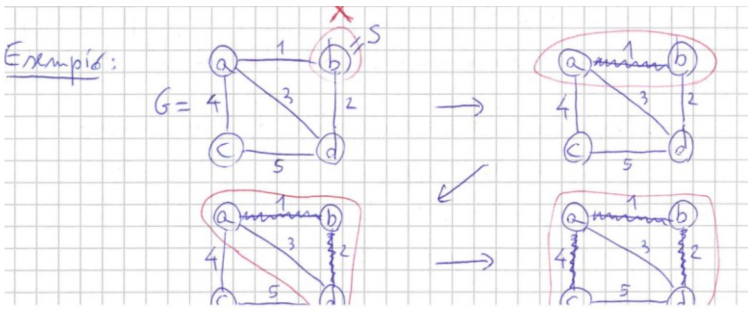
In summary, Prim's Algorithm operates by iteratively selecting safe edges, so light edges connecting the current tree with vertices not yet included in the tree. See the gif to see its run.

- Basically, it includes two sets of vertices
    - the ones already included in the MST
    - the other containing the ones not yet included
- At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these ones
- After picking that one, it moves the other endpoint of edge containing the MST

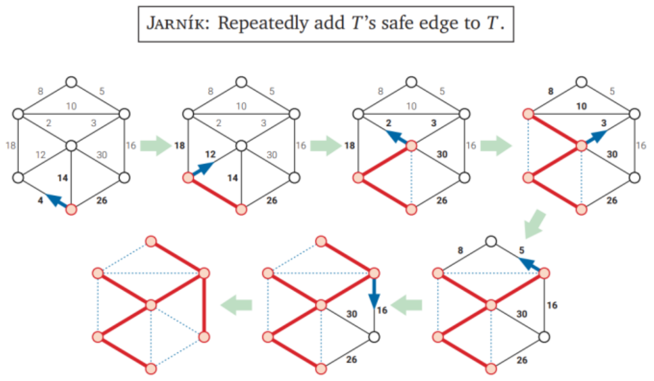
This algorithm "*grows*" a (spanning) tree from a source vertex s (doesn't matter who s is) by adding an edge that a time. This is implemented resulting in an efficient and optimal solution.

In the following example, a random vertex is chosen and then the algorithm is applied, following the principles discussed before in various points:



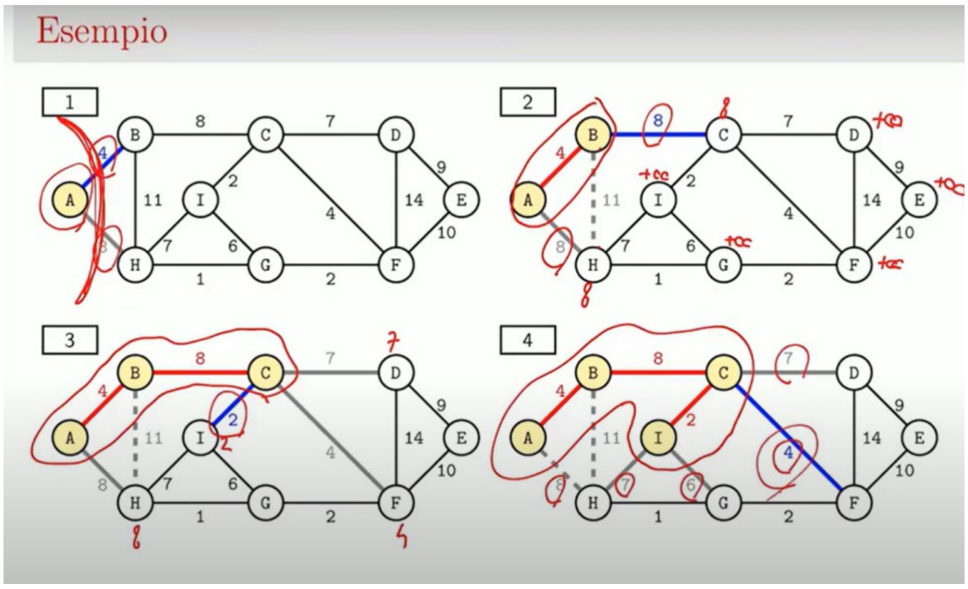


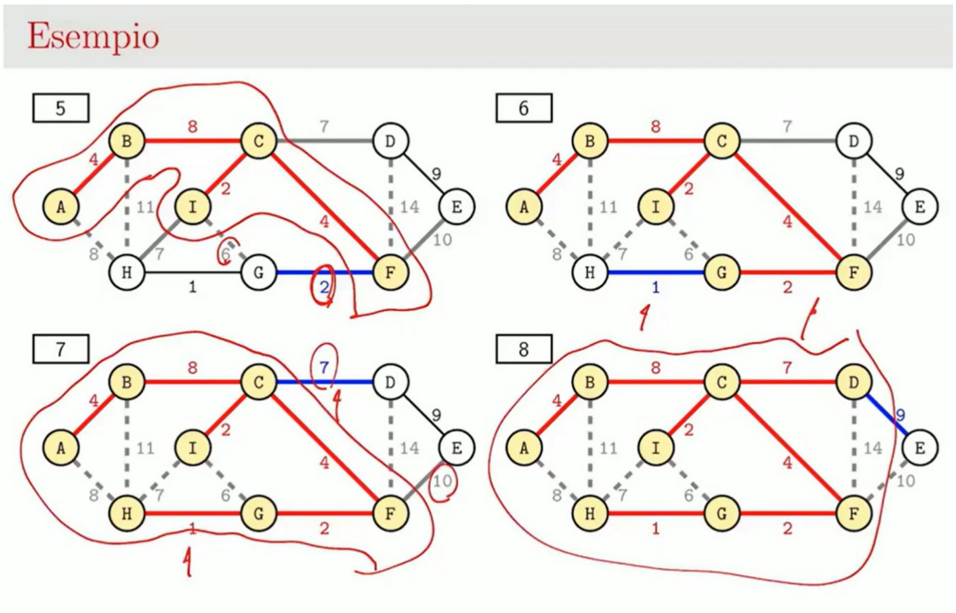
- *Correctness*: it follows from the theorem
- *Complexity*: $O(m * n)$ – depends on how it's *implemented*
    - Step $(u^*, v^*) = light\ edge$ was written too much sparsely to understand its complexity
    - n iterations doing at most m vertices
    - Assuming G is represented with an adjacency list
    - Keeping track of which edges are in X associating a boolean variable for each edge
    - In each iteration we do an exhaustive research on the edges to find the light edge
        - $O(m)$
    - There are $n - 1$ iterations, looking at all m edges to find the light edge
        - $O(m * n)$
    - Polynomial time → efficient algorithm (for small graphs)
        - This time is not efficient in practice (big/large/very large graphs)
            - Think of Facebook graph: $n \simeq 2\ billions * hundreds$
            - This means it's not so efficient in very large graphs

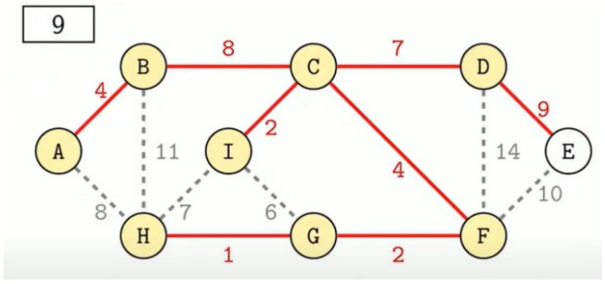


Written by Gabriel R.

A complete example just to make you really understand:







4.3.1 Efficient Prim's Algorithm – Heap implementation

Key observation: in the basic implementation, the computation of the light edge (minimum) is done *repeatedly* via exhaustive research (brute force= ⇒ should speed it up, so the algorithm and the computation will be faster.

<u>Golden rule</u> in algorithms/coding: when an algorithm repeats frequently the same question, look for the "right" data structure to speed that operation up. The right kind of data structure to do this is a *priority queue*, implemented with a <u>heap</u>.

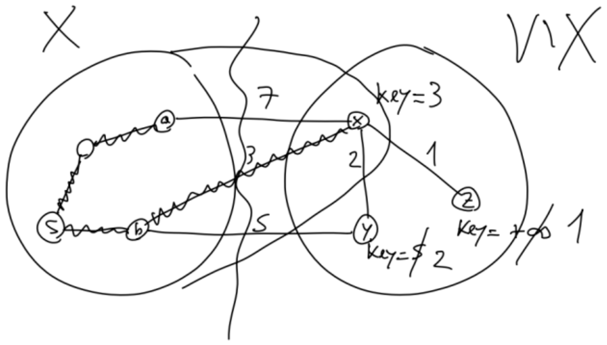
Recap about this data structure, to see which operations it allows:

- $insert$ → *add* an object to the heap (possibly fast)
- $extractMin$ → *remove* an object with the *smallest key* (highest priority)
  - to find the minimum, it uses in $O(1)$ time the operation min
- $delete$ → given a pointer to an object, *remove* it
- *Note*: In a heap with n objects, the complexity of these operations is $O(log(n))$

We can redefine the Prim's algorithm exploiting this efficient data structure, basically with the same principle; consider a min heap starting from whatever vertex, which is the root. From there, always extract the minimum value (means checking if it is min heap), then update the path.

Note: usually, this uses a priority queue Q, at least even in old years was like this here too.

Below figure shows that:

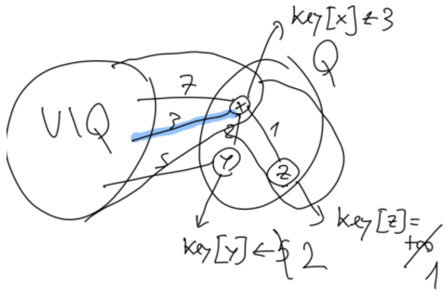


It's simple to store *vertices* in the heap (instead of edges) – whatever operation, the heap property will be established again, restoring keys and values. We can redefine Prim's implementation with a heap:

```
procedure $Prim\ (G, s)$            $//\ s = source\ vertex$
$//\ the\ heap\ will\ contain\ all\ vertices\ initially$
    for each $v \in V$: do
        $key[u] = +\infty\ //\min weight\ of\ any\ edge\ connecting\ v\ to\ the\ tree$
        $\pi(v) = NULL\ //\ \pi\ = parent\ of\ v\ in\ the\ tree\ being\ built$
$Key[s] = 0\ //\ set\ it\ to\ 0, because\ it's\ the\ source\ vertex$
$H = V\ //\ at\ the\ very\ beginning, the\ heap\ contains\ all\ vertices\ not\ in\ the\ tree$
while $H \neq 0$ do: $//while\ the\ heap\ is\ not\ empty$
```

```
    $v^* = extractMin(H)$ // $U$ is incident for a light edge for $(V \setminus H, H)$
    for each $v$ adjacent to $v^*$: do //only the adjacent nodes need to be updated and take the edge with lightest weight
    // starting from source, each other vertex will have weight $+\infty$ and we take all vertices not inside heap
      // update key and $\pi$ for each vertex adjacent to $v^*$ but not in the tree
      if $v \in H$ and $w(v^*, v) < Key(v)$ then
            $\pi(v) = v^*$ // $Key(v) = w(v^*, v) =$ including $v$ inside
            delete $v$ from $H$
            $Key(v) = w(v^*, v)$ //updating the heap with weight connecting $v$ to the tree, so $(u, v)$
            insert $v$ into $H$
```

We consider, during algorithm execution implicitly, we have $A = \{(v, \pi(v)) : v \in V \setminus \{s\} \setminus \{H\}\}$ (which means A keeps track of edges included and H becomes empty).



Complexity:

- $init$ → $O(n)$
- $while$ → n iterations
- $extractMin$ → $O(\log(n))$

Total cost of only $extractMin$ operations: $O(n\log(n))$

- `for` *loop*: executed $O(m)$ times in total (every vertex is explored) → $\sum_v \deg(v) = 2m$
  - $v \in H \to O(1)$
    - This here is a simple check
  - $Key(v) = w(v^*, v) \to delete + insert$: $O(2\log(n)) = O(\log(n))$
    - Two operations

Total cost of `for` *loop*: $O(m\log(n))$ (iterating for all adjacent nodes, quantity equal to node degree)

This way, the total complexity of the algorithm is $O(n\log(n) + m\log(n)) = O(m\log(n))$ (since G is connected, we recall) → near-linear time

Note: using Fibonacci heaps (more efficient, since they do amortized operations) $\Rightarrow O(m + n\log(n))$

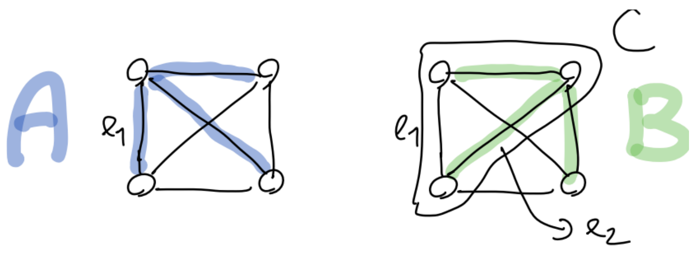
4.3.2 Exercises

Exercise (uniqueness of MSTs):

Show that if the weights of the edges are all distinct then there exists exactly one MST.

(*Hint: cut and paste argument – similar to the theorem correctness*)

Solution (with details of lesson but also other including Wikipedia and other sources)



Assume there are two MST different from each other, so the contrary and so $A \neq B \Rightarrow \exists$ an edge in one but not in the other; since weights are distinct, $\exists!$ with min weight, call it e_1, without loss of generality (not introducing any artificial assumption), $e_1 \in A$ and the argument is a cut-and-paste one (this choice will be unique, considering edge weights are all distinct from each other):

- Add e_1 to $B \Rightarrow$ this creates a cycle C; A is (M)ST $\Rightarrow A$ has no cycles $\Rightarrow C$ has an edge $e_2 \notin A, \neq e_1 \Rightarrow w(e_2) > w(e_1)$
    - Because e_1 was chosen as the unique lowest-weight edge (only edge with minimum weight not in the other) among those belonging to exactly one of A and B
    - Therefore the weight of e_2 must be greater than the weight of e_1
    - Now, both e_1 and e_2 are in C
- Remove e_2 from $B \Rightarrow$ get a new spanning tree with weight $< w(B)$ (so, smaller weight): contradiction, because B is an MST!

Two conclusions can be done:

- More generally, if the edge weights are not all distinct then only the (multi-)set of weights in minimum spanning trees is certain to be unique; it is the same for all minimum spanning trees
- *When the edge weights are not all distinct, it's possible for multiple different MSTs to exist*
    - however, while the actual arrangement of edges in these MSTs may vary, the set of weights of the edges across all MSTs will remain the same

Other exercises

1) *Is the converse true? (e.g., are weights necessarily unique for every possible graph and this has to hold for every graph)*

Solution

No: think of G as a tree (literally only thing professor will write – lame, I know, I added more).

A connected graph with repeated edge weights and this can still have a unique minimum spanning tree. Considered the trivial example of G being a tree; in this case, there are no cycles, so any spanning tree will be minimal, hence unique, regardless of repeated edge weights.

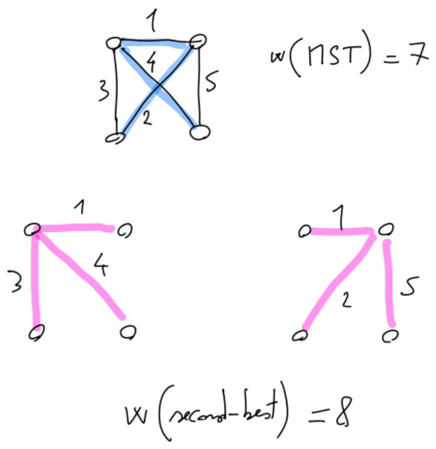
In conclusion, we might say:

- Distinct weights guarantee a unique MST
- Repeated weights can have multiple MSTs
    - But the set of weights used will always be the same across all of them

2) *Show that the <u>second best</u> MST, that is, the spanning tree of second-smallest total weight, is not necessarily unique (here we look for only one graph)*

<u>Solution</u>

There will be a unique MST, but for the second best, according to where the cut will be displaced, there will definitely be more than one, given the cut can be done on more than two edges at a time. In the following figure, there is only one MST of weight 7, but two second-best MSTs of weight 8.



If you want a complete formal explanation, see the book solution to this exercise here (look for problem B in the link).

4.4 KRUSKAL'S ALGORITHM

Another very simple, very famous and fast algorithm for MST is <u>Kruskal</u>, which dates back to 1956.

- It's as fast as Prim, both in theory and in practice (if properly implemented)
  - very quick and clean in its implementation
- It gives us the opportunity to study a new data structure

It picks the minimum weighted edge at first and the maximum weighted edge at last. It sorts edges by weight and then adds them continuously, preserving the "safe edge" property – take only the unexplored. It does so preventing the adding of cycles.

It implements the $Generic - MST(G)$ algorithm, but:

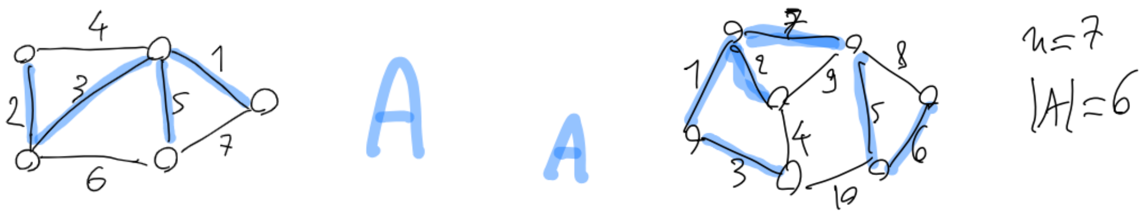
- A is a forest (set of trees)
- Safe edge is a light edge connecting 2 distinct components

Here goes the *pseudocode*:

```
procedure $Kruskal(G)$ $\backslash\backslash$ $no\ source\ vertex\ needed$
    $A = \emptyset$
    $Sort\ safe\ edges\ of\ G\ by\ weight$    $//\ e.g. use\ MergeSort$
    for each $edge\ e\ in\ non-decreasing\ order\ of\ weight$: do $//so, order\ is\ ascending$
        if $A \cup \{e\}\ is\ acyclic$ then: $//\ if\ I\ can\ add\ this\ new\ edge\ without\ creating\ a\ cycle$
```

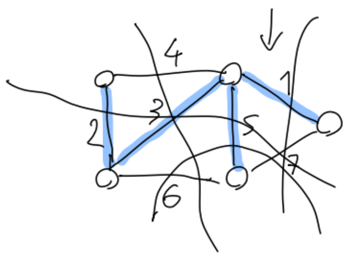
```
            $A = A \cup \{e\}$ // go ahead and add it
    return $A$
```

Consider the following examples – vertices will be sorted by weight and the result is not a tree but a set of two trees, adding only ones that do not form a cycle:

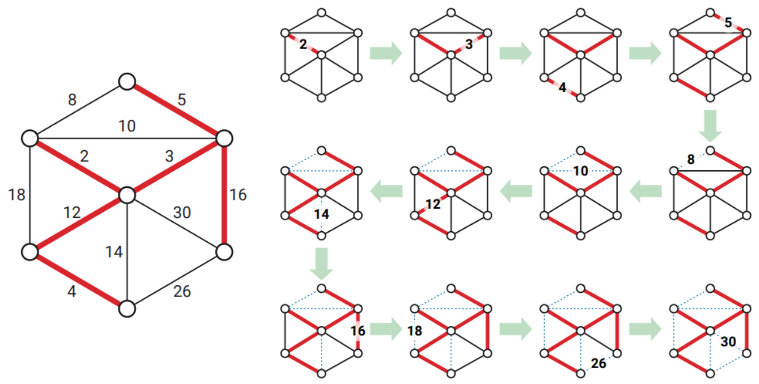


Simple/small optimization of the algorithm: stop the *for loop* when A has $n - 1$ edges. Note also that a source vertex is not needed here.

Correctness: follows from correctness of $Generic - MST$ (here – basically, it's always the same algorithm, only seen in a different way, considering it's always "respecting" cuts in a safe way, as shown in next figure).



KRUSKAL: Scan all edges by increasing weight; if an edge is safe, add it to F.



Complexity:

- Sorting: $O(m \log(n))$
- For loop: check whether $e = (u, v)$ closes a cycle is equivalent to check whether A contains an $u - v$ path → DFS/BFS in linear time on $G = (V, A)$, which has n vertices and $\leq n - 1$ edges
- complexity: $O(n)$

Total: $O(m * n)$ → $O(\cancel{m \log(n)}) + O(m * n) = O(m * n)$ [because implemented with adjacency list]

Question: *Can we implement it faster?* Let's delve into it in the next subsection.

4.4.1 Efficient Kruskal

It can be implemented as fast as Prim's, considering only frequent operations done inside the previous algorithms.

- Kruskal frequent operation = cycle check (equivalently, path check)
    - It happens when an edge is added to A

No data structure seen up until now is able to do this. We use a new data structure supporting this operation fast; it's called Union-Find (also called *disjoint set*), created in 1964.

This is a structure to merge *disjoint sets* (also non-overlapping in their elements) of objects and supports at least three operations:

- *Init* (or *Initialize*): given an array X of objects
    - it creates a Union-Find data structure with each object $x \in X$ in its own set
- *Find*: given an object x, return the name of the set that contains x
- *Union*: given two objects x, y merge the sets that contain x and y into a single set
    - done whenever the sets are distinct
    - if x, y are already in the same set, this operation does nothing

These operations can be implemented with the following complexities:

- *Init*: $O(n)$
- *Find*: $O(\log(n))$
- *Union*: $O(\log(n))$

(*Note*: here, we assume n represents the number of objects in the data structure)

We want to implement Kruskal fast using the Union-Find structure:

- Idea: Union-Find (U-F) keeps track of the connected components of the current solution
    - whenever I need to check if the merge unites with existing edges
    - $A \cup \{(v, w)\}$ creates a cycle $\Leftrightarrow$ v, w are already in the same connected components

```
procedure $Kruskal(G)$
    $A = \emptyset$
    $U = init(V)$                    $// U - F\ data\ structure$
    $sort\ edges\ of\ E\ by\ weight\ // e.g. using\ Merge\ Sort$
    for each $edge\ e = (v, w)\ in\ non-decreasing\ order\ of\ weight$: do
    if $Find(v) \neq Find(w)$ then: $// if\ values\ are\ different, they\ can\ be\ added\ within\ union$
        $// no\ v - w\ path\ in\ A, so\ OK\ to\ add\ e$
        $A = A \cup \{(v, w)\}$
        $// update\ due\ to\ component\ union$
        $Union(v, w)$
return $A$
```

Written by Gabriel R.

Complexity:

- Init: $O(n)$
    - actually, before the loop it's $O(n) + O\big(mlog(n)\big)$
- Sorting: $O(m\log(n))$
- $2m$ Find: $O(m\log(n))$
- $n-1$ Union: $O(n\log(n))$ → only when I go inside an "if" and when the edge is added
- A updating: $O(n)$
    - basically, the remaining part

In total: $O(m\log(n))$

4.4.2 Union-find Implementation

(Further readings for this part: here, here and here)

We'll use an array, which can be visualized as a set of directed trees. Each element of the array has a field $parent(x)$ that contains the index of the array of some object y.

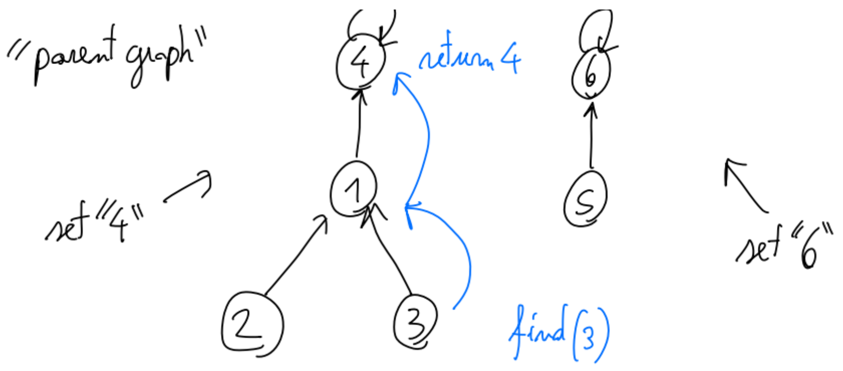
Consider the following example:

index of x	$parent(x)$
1	4
2	1
3	1
4	4
5	6
6	6

You can see the previous as:

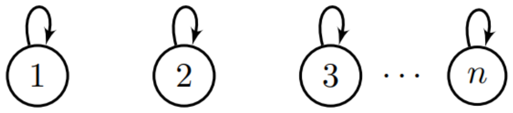
- Vertices: (indexes of) objects
    - called by index inside of the array
- Edge $(x, y) \Leftrightarrow parent(x) = y$
    - basically, direct edge

A set of object corresponds with a set of trees directed to the "parent graph":



As convention: name of the set = root of that tree. We are giving the names of the roots, hence why we call them *set* "4" and *set* "6", visualized with direct trees. Consider the following operations:

- Init



Allows to initialize nodes as the parents of themselves (hence, the self-loops you see above), so more precisely $for\ each\ i = 1,2,\dots,n$, initialize $parent(i)$ to i

- Find

It follows parent by parent until the root is reached $parent \to parent \to \cdots \to root$. So basically, it returns the connected component of the vertex to search.

Consider the following procedure – finds the name of a set containing the node to find and returns it. Basically, goes up father by father reaching a root identifiable as a self-loop.

$Find(x)$:

```
1) Starting from $x$'s position in the array
    a. traverse parent edges until reaching a position $j$
    b. s.t. $parent(j) = j$
2) return $j$
```

From figure, an example: $Find(3) \Rightarrow\ 3 \to 1 \to 4\quad return\ 4$

Definition: The depth of an object x is the number of edges traversed by $Find(x)$

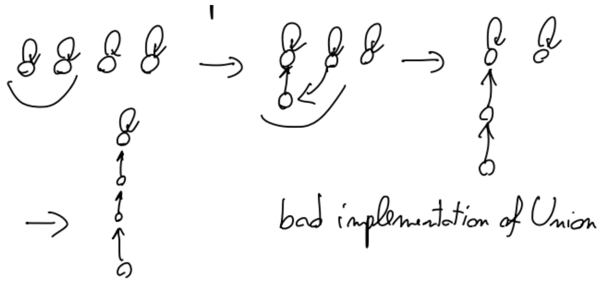
From figure, an example:

$depth(4) = depth(6) = 0, depth(1) = depth(5) = 1, depth(2) = depth(3) = 2$

The *complexity of* $Find(x)$ is proportional to the largest depth of x, depending on the Union implementation.

- In general, the worst case complexity is $O(n)$, depending on how the trees are merged
  - In any case, it strictly depends from the largest depth of an object/the biggest height of the parent graph
- In any case, the maximum height of a tree is $(n-1)$

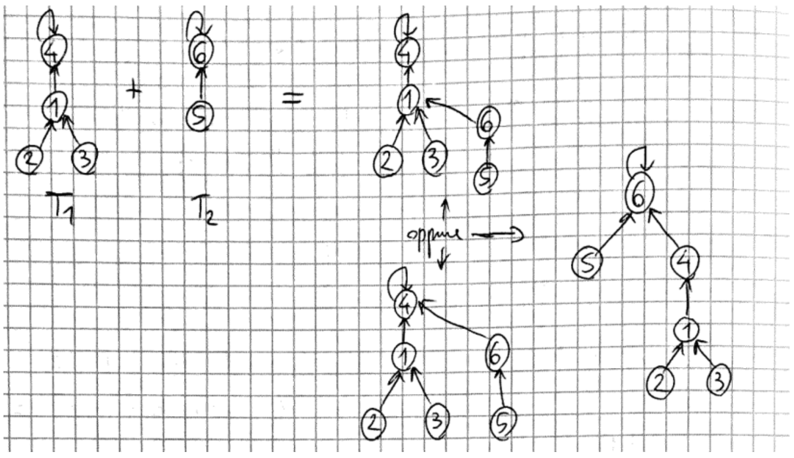
As shown in figure, this strictly depends on the specific case; some are better, some are worse:

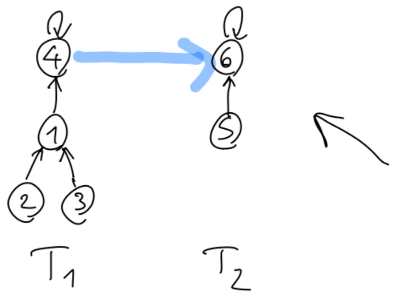


- Union

$Union(x, y) \rightarrow$ given two objects x, y

- The two trees of the parent graph containing x and y must be merged in a single tree
- The simplest way is to point one of the 2 roots to one *another node* of the other tree





We need to decide:

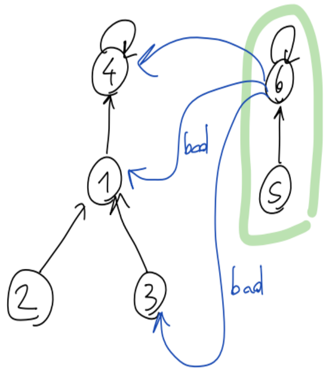
1) Which of the 2 roots remains a root
2) To which node should a root point (equivalently, where the new edge is directed)

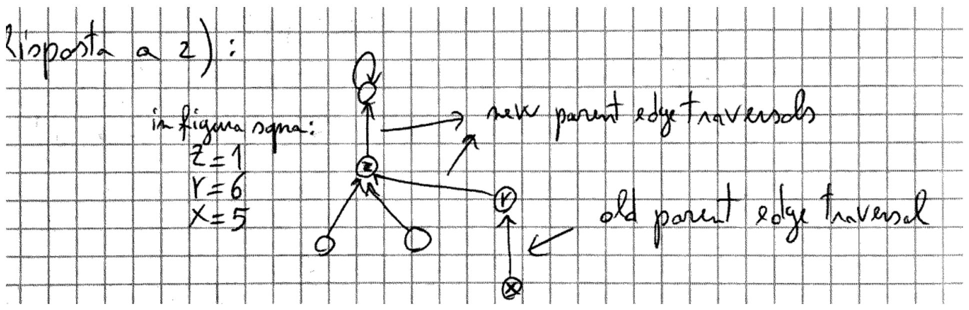
Up next, we try to answer to (2).

In the right figure (for Point 1) the root (4) is pointed, because we want to keep the depth as low as possible. That's the simplest thing that we can do.

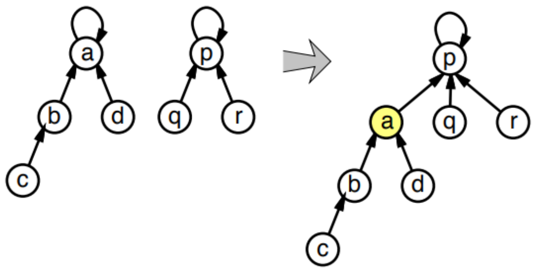
Every level below the root would make the depth bigger by 1 (because there's a new parent edge) and the depth of z.

(For Point 2) We would need to have (6) pointing (4), because otherwise depth would increase – minimum possible increasing depth – so do not take 1 or 3 but the root. So, the root has to point to the other root, so to increase depth only by 1.



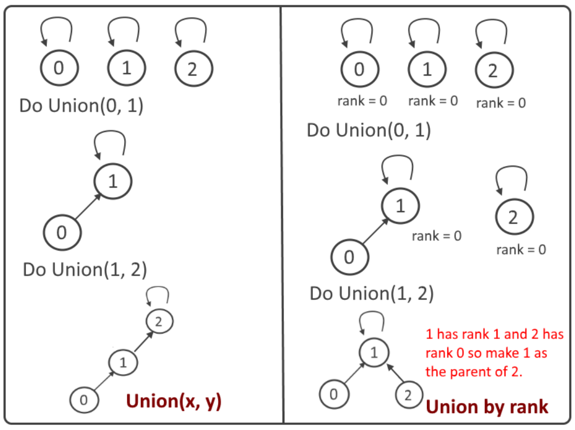


A better example of Union to clearly see what's going on:



We would need to have the smallest amount of nodes pointing to the biggest one – in other words, to have the least increasing depth, there are two alternatives, concerning Point 1:

- One possible idea for is *to minimize the number of objects* whose depth will increase
    - This way, objects depth increases as much as the depth of the tree in which it's been chosen to set the tree
    - This is called "<u>union-by-size</u>". This is the chosen option indeed.
- Alternatively: the root of the least tall tree points to the one of the tallest tree
    - This is called "<u>union-by-rank</u>"



To make you actually understand:

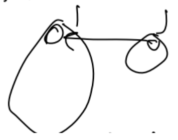
- *Union by Size* means that during the union operation, the root of the smaller tree (the one with fewer elements) is made a child of the root of the larger tree (the one with more elements)
- *Union by Rank* means that during the union operation, the root of the tree with a lower rank is made a child of the root of the tree with a higher rank
    - The rank is a rough estimate of the tree's height.

The <u>rank</u> of a node generally refers to the distance (the number of nodes including the leaf node) between the furthest leaf node and the current node. Basically rank includes all the nodes beneath the current node – so, it keeps track of the height or the depth.

For the procedure $Union(x, y)$:

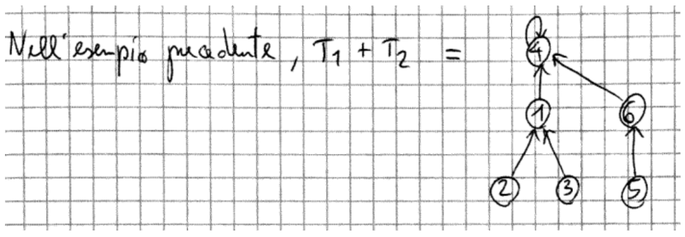
1) Invoke $Find(x)$ and $Find(y)$ to obtain the names i and j of the sets that contain x and y
    a. Basically, just to understand in which tree they are
    b. If $i = j$ then *return* (aka – there is nothing to do)

field to be added to the datastructure.



```
2) if $\underline{size(i)} \geq size(j)$ then
       $parent(j) = i$
       $size(i) = size(i) + size(j)$  // update the size of all objects present inside the tree rooted in j
   else
       $parent(i) = j$
       $size(j) = size(i) + size(j)$ // update the size of all objects present inside the tree rooted in i
```

Here, consider $size$ is a field to be added to the data structure apart from $parent(x)$ counting the elements in the tree with root x.



4.4.3 Exercises

Exercise

Argue that the complexity of $Find(x)$ (and of $Union(x, y)$) is $O(\log(n))$.

Solution

Initially, $depth(x) = 0 \ \forall x.\ depth(x)$ can only increase because of a Union in which the root of the tree of x points to another root (depth increases by 1 by construction). This happens only when the tree of x gets merged to a tree of size not smaller (at least as big) ⇒ when the depth of x increases, the size of the tree of x at least doubles.

For union by size, infact, when two trees are united, the tree with fewer nodes is attached to the root of the tree with more nodes.

- How many times can this happen?
    - Given the size of the tree is $\leq n$
    - $\leq \log_2 n$ times (at most)
        - Therefore the depth of x cannot increase more than $\log_2 n$ times
        - Depth increases by ≤ 1
        - $\Rightarrow depth(new\ tree)\ \ O(\log(n))$

So, we have two different algorithms with complexity $O(m \log(n))$. To reach complexity $O(m)$ is still an open problem (there are slightly faster algorithms, but not not others optimal able to reach $O(m)$). Given the complexity of $Find(x)$ is proportional to $depth(x)$, this is $O(\log(n))$.

Exercise (made in Italian years)

The *maximum spanning tree* of a graph is a spanning tree of maximum cost, of which the sum $\sum_{e \in T} w(e)$ is maximum. Give an algorithm for which this problem which uses as procedure an algorithm to solve minimum spanning tree problem.

Algorithm:

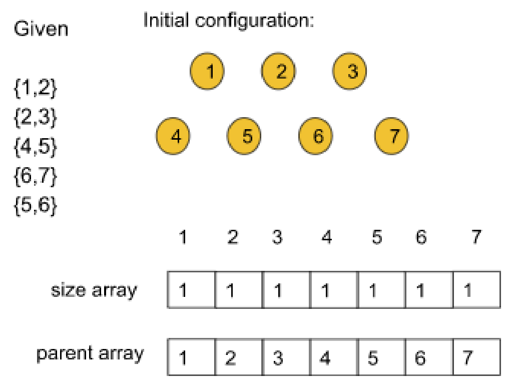
- Multiply by -1 the weights of all edges
- Apply Kruskal the algorithm

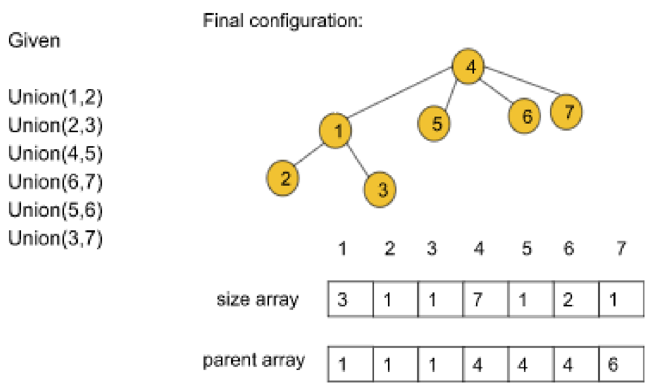
Super-fast algorithms for MST:

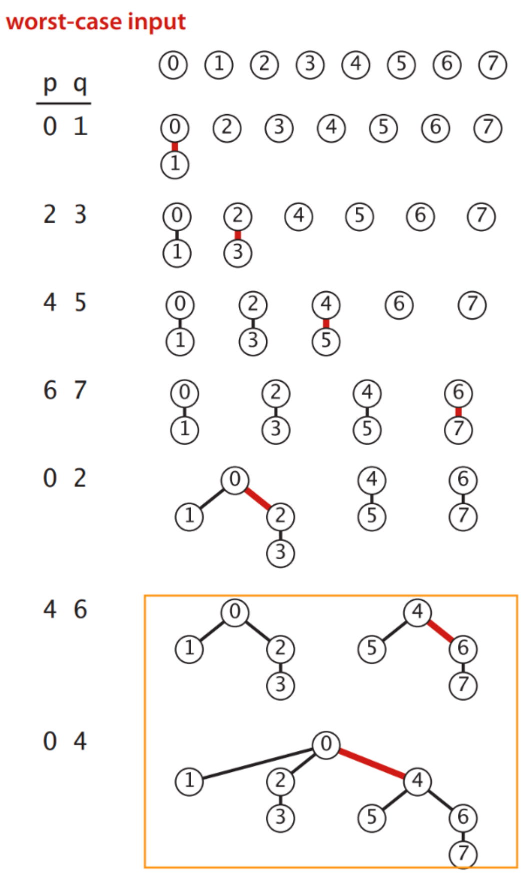
- Prim/Kruskal → $O(m\log(n))$
- Fredman-Tarjan (Prim + Fibonacci heaps – 1984) → $O(m + n\log(n))$
- Fredman-Tarjan (1984) → $O(m\log^{*}n)$ $(\log^{*} 2^{65536} = 5)$ → N. of atoms in the universe
- Gabow et al. (1986) → $O\big(m\log(\log^{*}(n))\big)$
- Chazelle (2000) → $O(m\,a^{m,n})$ $(a(2^{65536}, 2^{65536} \leq 4)$

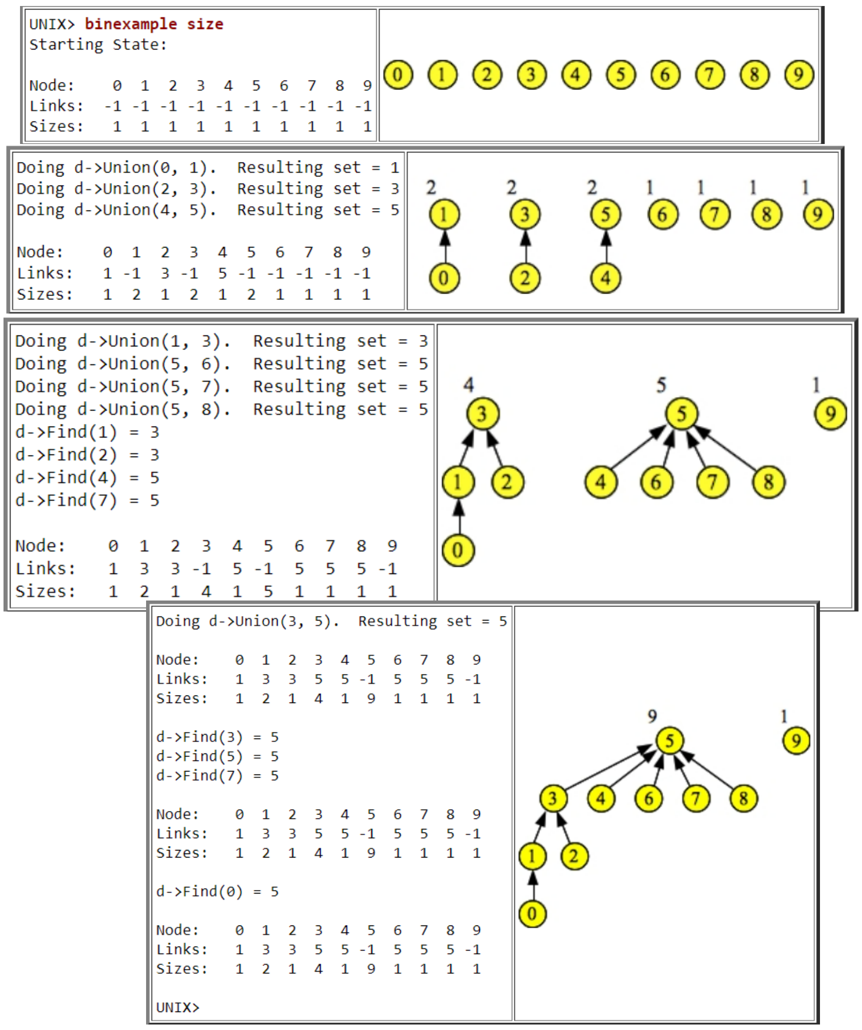
4.4.4 Examples of union by size (and others)

Given it was asked in an older exam, here we have an example of union by size (here you can find all):









```
UNIX> binexample size
Starting State:

Node:    0  1  2  3  4  5  6  7  8  9
Links:  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
Sizes:   1  1  1  1  1  1  1  1  1  1
```

```
Doing d->Union(0, 1).  Resulting set = 1
Doing d->Union(2, 3).  Resulting set = 3
Doing d->Union(4, 5).  Resulting set = 5

Node:    0  1  2  3  4  5  6  7  8  9
Links:   1 -1  3 -1  5 -1 -1 -1 -1 -1
Sizes:   1  2  1  2  1  2  1  1  1  1
```

```
Doing d->Union(1, 3).  Resulting set = 3
Doing d->Union(5, 6).  Resulting set = 5
Doing d->Union(5, 7).  Resulting set = 5
Doing d->Union(5, 8).  Resulting set = 5
d->Find(1) = 3
d->Find(2) = 3
d->Find(4) = 5
d->Find(7) = 5

Node:    0  1  2  3  4  5  6  7  8  9
Links:   1  3  3 -1  5 -1  5  5  5 -1
Sizes:   1  2  1  4  1  5  1  1  1  1
```

```
Doing d->Union(3, 5).  Resulting set = 5

Node:    0  1  2  3  4  5  6  7  8  9
Links:   1  3  3  5  5 -1  5  5  5 -1
Sizes:   1  2  1  4  1  9  1  1  1  1

d->Find(3) = 5
d->Find(5) = 5
d->Find(7) = 5

Node:    0  1  2  3  4  5  6  7  8  9
Links:   1  3  3  5  5 -1  5  5  5 -1
Sizes:   1  2  1  4  1  9  1  1  1  1

d->Find(0) = 5

Node:    0  1  2  3  4  5  6  7  8  9
Links:   1  3  3  5  5 -1  5  5  5 -1
Sizes:   1  2  1  4  1  9  1  1  1  1

UNIX>
```

5 Shortest Path

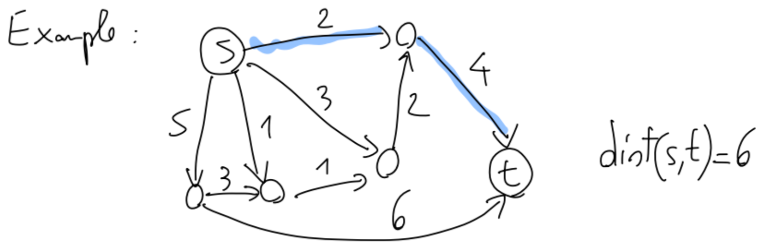
5.1 Definition and Terminology

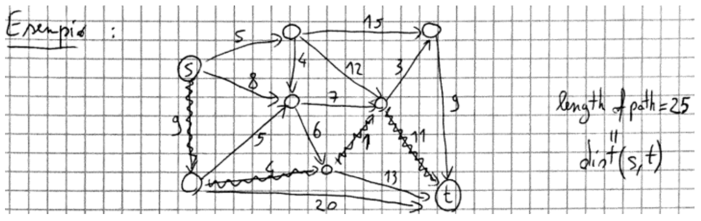
We give some definitions:

- Given a weighted and directed graph, the length of a path $p = v_1, v_2, \dots v_k$ is defined as $len(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$ (basically, it's a sum of all the weights for this path)
- A shortest path from a vertex u to a vertex v is a path with min. length among all $u - v$ paths
- The distance between 2 vertices s and t, denoted as $dist(s,t)$ is the length of a shortest path from s to t; if there is no path at all from s to t then $dist(s,t) = +\infty$

5.2 Problem and Application

Given a directed, weighted graph and a source vertex $s \in V$ and a destination $t \in V$, compute the shortest path from s to v. Consider the following example of a generic graph:





Observation: in directed graphs, in general $dist(u,v) \neq dist(v,u)$

Applications:

- Road networks (Google Maps)
- Routing in networks (e.g., Internet)
- Robots navigation

Instead of solving the problem just presented, we will solve it using a different one.

5.3 Single-Source Shortest Paths (SSSP)

Definition:

- *Input*: A directed, weighted graph $G = (V, E)$ with source vertex $s \in V$ with length $l_e \geq 0$ with edge weights $w: E \to \mathbb{R}$ and a source vertex $s \in V$
- *Output*: $dist(s,v), \forall v \in V$ → total length of the shortest path from s and v
    - Shortest path to all destinations generally

Comments:

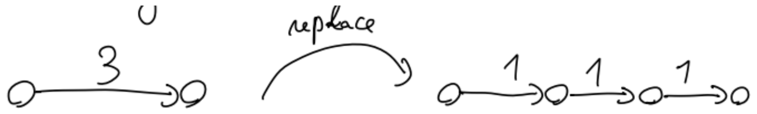
- No algorithms are known for the previous problem that run asymptotically faster than the best SSSP algorithm in the worst case scenario

- We'll work with directed graphs
    - Because it's more general in nature
        - So if we work with these, also we can with undirected
        - With few "cosmetic" modifications
    - But all the algorithms that we'll see can be adapted easily for undirected graphs
- The assumption $l_e \geq 0 \; \forall e \in E$ is meaningful. It's true in a lot of applications (distances between roads), but not in all!

5.3.1 Non-negative edge weights

We'll first solve a special case: non-negative edge weights $w: E \rightarrow \mathbb{R}_{\geq 0}$. We've already solved a special case of this one, which was when all weights are equal (specifically, when all the weights are $w = 1$) → already solved it in linear time using BFS.

Consider the following idea: an edge of weight different from 1 (can be $w = n$), but we force it to be using a *replace* with all edges with weight 1 (replaced with n edges with weight $w = 1$).



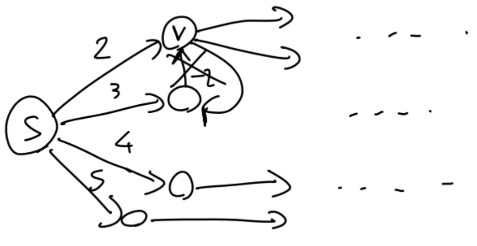
This one uses BFS (with *reduction* – aka "use a problem to solve another one") – the reduction step is important because BFS allows to compute shortest paths intended as number of edges of the path, counting each one as single-weighted (we take a "longer" edge and treat it as "n" edges with 1 as value).

- *First issue*: integer weights, so it's a still special case (not general)
- *Second (bigger) issue*: the size of the graph can be much bigger than the size of the starting graph; the length of the graph can be $\gg n, m$)
    - ⇒ BFS takes linear time in the "bigger" graph
        - and this is not necessarily linear time in the size of the original graph
        - this can definitely grow exponentially

5.3.1.1 Intuition of a new algorithm

We consider a source vertex s and the problem of finding a shortest path towards all of them.

- This continues recursively, between nodes, arcs and other arcs, continuing each time
- Apart from the shortest path we see, can there be another shortest path?
    - We can't because there aren't negative edges
    - Any other way, we pay more



The edge (s, v) must be the shortest path from s to v since the first segment of any other path is already larger, and the weights are non-negative; a similar reasoning works in the next steps.

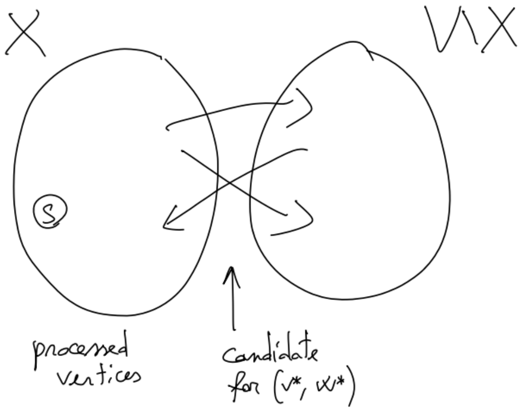
5.3.2 Dijkstra's algorithm

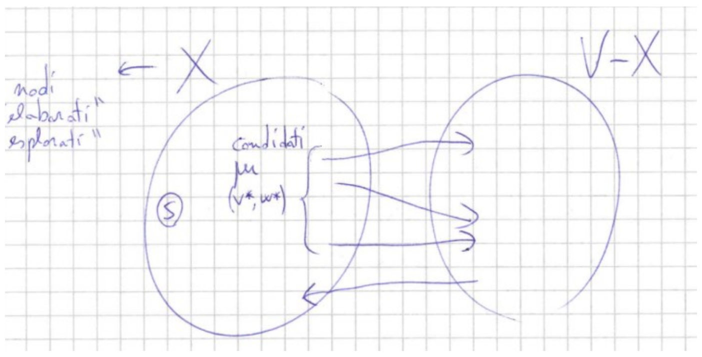
Probably one of the most famous algorithms in Computer Science, born in 1956. This is a greedy algorithm, very similar to Prim.

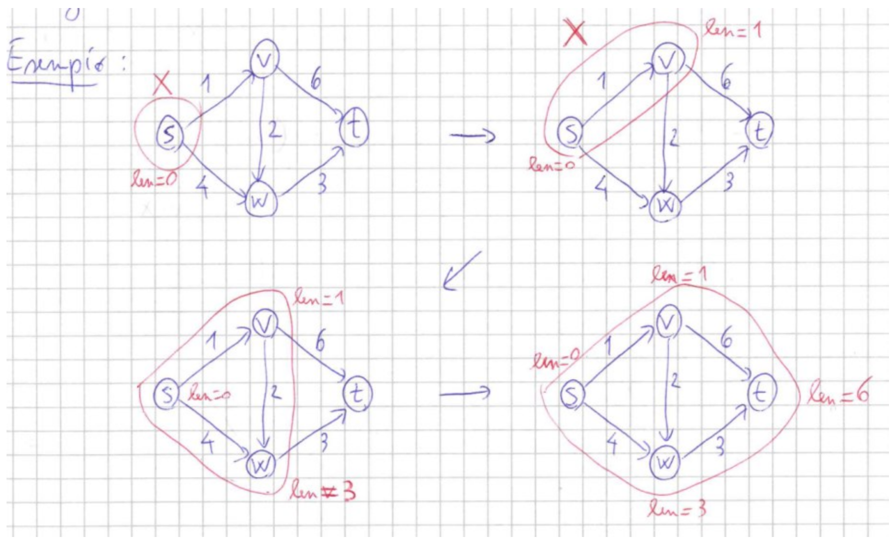
- *Input*: directed G as adjacency list, $l_e \geq 0, \forall e \in E, s \in V, w: E \to \mathbb{R}_{\geq 0}$
- *Output*: $dist(s,v) = len(v), \forall v \in V$
  - With $len(v)$ coming as shorthand form of the previous one

```
procedure $Dijkstra(G,s)$
    $X = \{s\}$ // set of vertices which have been processed so far – we already know the shortest path to them
    $len(s) = 0$ //shortest known distance from source vertex
    $len(v) = \infty$ // initial estimated distance
    while there is an edge $(v,w)$ with $v \in X$ and $w \notin X$: do
        $(v^*, w^*)$ = such an edge minimizing $len(v) + w(v,w)$
        add $w^*$ to $X$ // add destination vertex to the previous set
        $len(w^*) = len(v^*) + w(v^*, w^*)$ //updates the shortest known distance
    // the length found to be the minimum is the definitive shortest path length for that new vertex
```

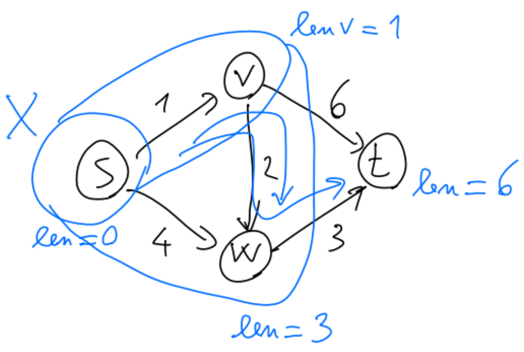
Consider the following figure – here, we iterate and each time we process a new node w^*:



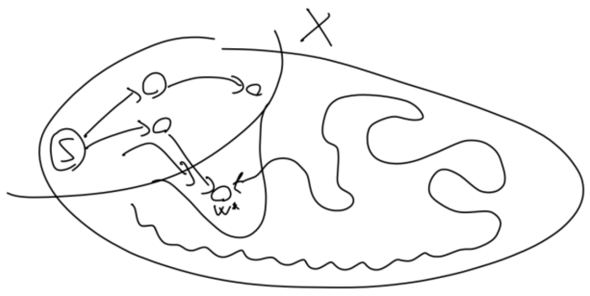




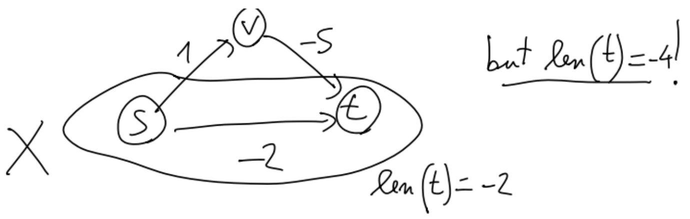
We do a very quick example – a set of vertices grows starting from a source vertex. We find an edge minimizes the length of the shortest path. Each time, a light path is taken incrementally.

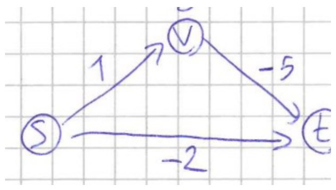


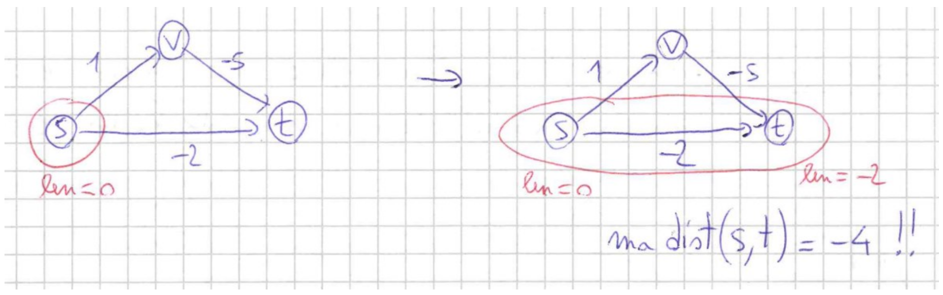
Praise of Dijkstra's algorithm: in each iteration, it <u>irrevocably</u> and myopically estimates the shortest path distance to an additional vertex despite having so far looked at only a <u>fraction</u> of the graph!



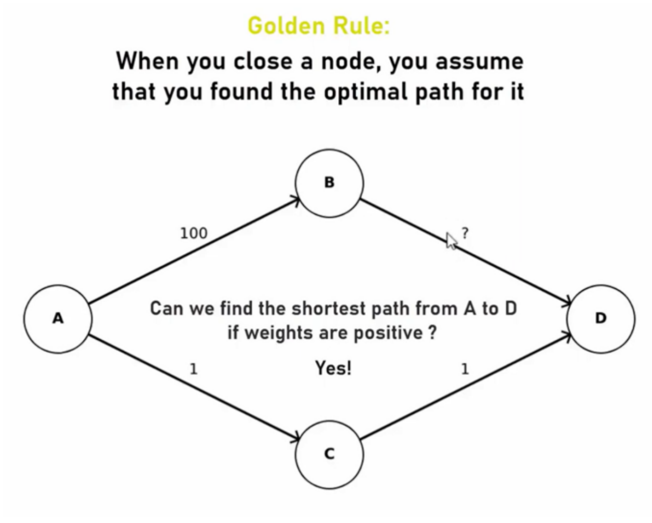
Observation: Dijkstra's algorithm does <u>not</u> work on graphs with negative weights



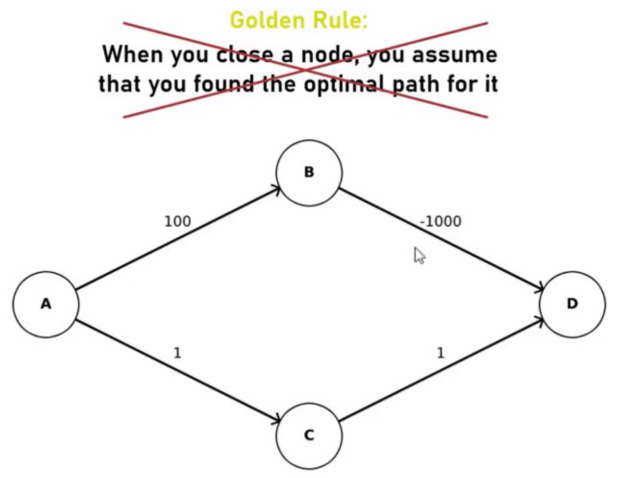




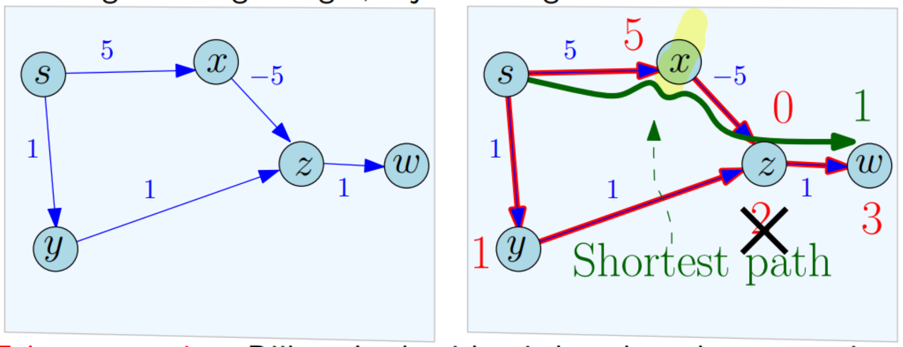
This means simply the following: here, all weights are positive, and the shortest path is actually the bottom one, no matter what value we will put inside the question mark.


Golden Rule:
When you close a node, you assume that you found the optimal path for it
A
B
C
D
100
?
1
1
Can we find the shortest path from A to D if weights are positive ?
Yes!

Here, instead, the optimal path is the upper one, but here, it won't even be considered, and we also know Dijkstra does not revise its decisions.


Golden Rule:
~~When you close a node, you assume that you found the optimal path for it~~
A
B
C
D
100
-1000
1
1

With negative length edges, Dijkstra's algorithm can fail

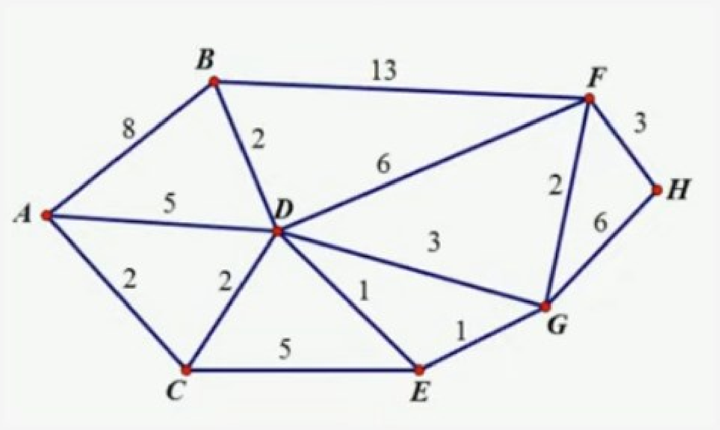


False assumption: Dijkstra's algorithm is based on the assumption that if $s = v_0 \to v_1 \to v_2 \ldots \to v_k$ is a shortest path from s to v_k then $dist(s, v_i) \le dist(s, v_{i+1})$ for $0 \le i < k$. Holds true only for non-negative edge lengths.

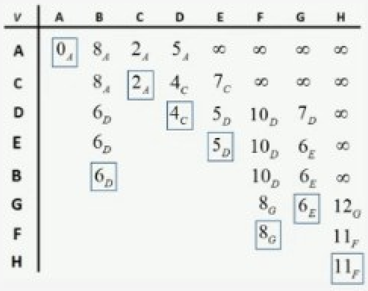
$d(s, x) > d(s, z)$

Another example:

The algorithm start from the source and update the weights of the path going out of it. Then it repeats the algorithm taking as source the vertex that is lightest reached. E.g.



In the iteration table the element in a blue box are the element in which the algorithm iterates, in fact they are the element for which there can't be a shortest path in the following iteration



V	A	B	C	D	E	F	G	H
A	$\boxed{0_A}$	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	$\boxed{2_A}$	4_C	7_C	∞	∞	∞
D		6_D		$\boxed{4_C}$	5_D	10_D	7_D	∞
E		6_D			$\boxed{5_D}$	10_D	6_E	∞
B		$\boxed{6_D}$				10_D	6_E	∞
G						8_G	$\boxed{6_E}$	12_G
F						$\boxed{8_G}$		11_F
H								$\boxed{11_F}$

Consider for the exam some examples are present in the relative chapter.

5.3.2.1 Correctness and complexity

Complexity of Dijkstra's algorithm: $O(m * n)$

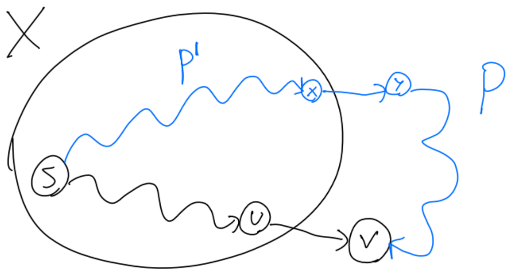
Correctness of Dijkstra's algorithm:

- Invariant: $\forall x \in X, len(x)$ is $dist(s, x)$ the length of minimal path from s to x

The proof goes on by induction on $|X|$:

- Base case: $|x| = 1$
  - Trivial, only the source vertex (path to 0)

- Inductive hypothesis:
  - Invariant is true $\forall |x| = k \geq 1$ $\exists w^*$
  - Let v the next vertex added to X and (u, v) the arc $\rightarrow (v^*, w^*)$
  - The shortest path s to $u + (u, v)$ is a path s to v of length $\pi(v) = \min_{(u,v): u \in X, v \notin X} len(u) + w(u, v)$
  - Consider any path from p from s to v we'll show that P is not shorter than $\pi(v)$: (basically, any vertex crossing X)



Let (x, y) be the first arc in P that traverses X and let P' be the sub-path from s to x.

Since edges are non-negative, the length P is non-negative and at least the length of P' plus the weight of that edge:

$$len(P) \geq len(P') + w(x, y)$$

$\downarrow$ w is nonnegative

$$\geq len(x) + w(x, y) \geq$$

↳ ind. hypothesis

$$\geq \pi(y)$$

↳ def. of π

$$\geq \pi(v)$$

↳ Dijkstra selected v instead of y

(Basically, any possible path can be selected, but only the shortest ones are selected, and this is correct since the choice is safe already. Any possible path is not shorter than Dijkstra's selection, given it's already the shortest possible. It only holds for non-negative edges, remember).

Written by Gabriel R.

5.3.2.2 Exercises

<u>Exercise</u>

Write an implementation of Dijkstra's algorithm with heaps.

<u>Solution</u>

```
procedure $Dijkstra(G, s)$        (almost identical to Prim's implementation with heaps)
    $X = \{s\}$
    $H = \emptyset$ // initialize heap
    $key(s) = 0$ // initialize key of source vertex
    for each $v \neq s$: do // iterate for all vertices
            $key(v) = \infty$
m
    for each $v \in V$: do // insert inside min heap
            insert $v$ into $H$
    while $H$ is non − empty: do // check inside all of the heap
            $w^* = extractMin(H)$
            add $w^*$ to $X$
            $len(w^*) = key(w^*)$
            // Update the heap
            for each edge $(w^*, y)$ s.t. $y \notin X$: do
                    delete $y$ from $H$
n                   $key(y) = \min\{key(y), len(w^*) + w(v^*, w^*)\}$
                    insert $y$ into $H$
```

(This algorithm gives only the length of the path, but it's not difficult to also insert the actual path inside of this one)

Complexity:

- considering the graph as adjacency list, n vertices and m edges
- $\log(n)$ iterations because of heap usage

Total number of operations: $O((n + m)\log(n))$

(there are $O(m + n)$ operations on heaps and each one has complexity $O(\log(n))$)

Another exercise, done in previous years:

Exercise: when ∃! shortest path from s to t:

a) All lengths are integer and distinct

b) All lengths are distinct powers of 2

c) (a) + there is no directed cycle

Answer: (b) → 2 sums of distinct powers of 2 cannot be the same number

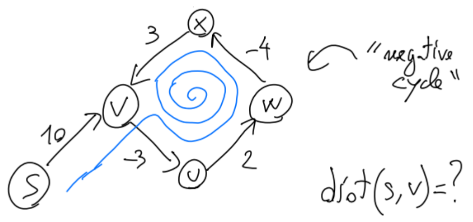
5.4 General SSSP Problem

(Further readings: here and here)

Let's look back at the general case, not the special one anymore. In this one, graphs can have edges with negative weights. Who cares about negative weights?

1) In road networks, traversing one edge comes with a reward/bonus → weights represent more general costs than just distance
2) Compute a profitable sequence of financial transactions

With negative weights we must be careful about what we even mean by "shortest paths". We ask ourselves what's the shortest path between s and v. It's also seen we have a negative cycle, keeping going inside of the graph indefinitely.

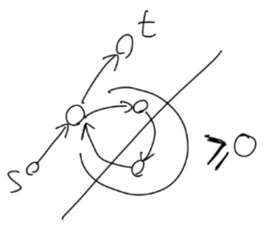


We conclude there is no shortest $s - v$ path ⇒ $dist(s, v)$ is undefined ($\infty, -\infty$). So, how about forbidding negative cycles (that is, compute shortest cycle-free/simple paths). Now the problem is well-defined, but it's NP-Hard (problem for which we don't have any polynomial time algorithm, unless $P = NP$).

5.4.1 Single-Source Shortest Paths (revised version)

- *Input*: Directed weighted graph $G = (V, E)$ and a source vertex $s \in V$
- *Output*: One of the following:
  - $dist(s, v)$ ∀ vertex $v \in V$
  - A declaration that G contains a negative cycle

Observation: can a shortest path contain a cycle? Not negative-weight cycles, but not positive-weight either. This is very simple to see, considering we enter a loop in which cost is always bigger.

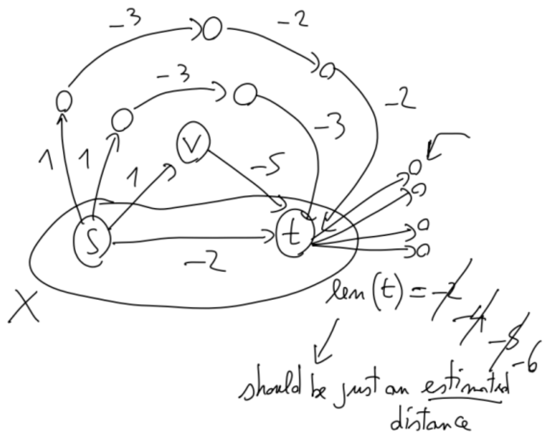


Written by Gabriel R.

What about 0-weight cycles? We can remove all of them, therefore wlog we can assume to compute cycle-free shortest paths, which have (at most) $\leq n-1$ edges.

What needs to be changed in Dijkstra's algorithm in order to make it able to deal with negative-weights edges?

- Consider an example below: we take the shortest path each time between vertices
  - It continuously updates its set adding a new vertex
- Every time a new vertex gets added, it never comes back to its decision



-3
-2
-3
-2
-3
1
1
1
-5
v
s
t
-2
X
len(t) = ~~-2~~ ~~-4~~ ~~-8~~ -6
should be just an estimated distance

Note that this is important, because in each relaxation step, the algorithm assumes the "cost" to the "closed" nodes is indeed minimal, and thus the node that will next be selected is also minimal.

The idea of it is: If we have a vertex in open such that its cost is minimal - by adding any positive number to any vertex - the minimality will never change.

Without the constraint on positive numbers, *the above assumption is not true*.

Since we do "know" each vertex which was "closed" is minimal - we can safely do the relaxation step - without "looking back". If we do need to "look back" - Bellman-Ford offers a recursive-like (DP) solution of doing so.

Problem:

- It never revisits/updates its decisions, but it should for all vertices!
  - Once a vertex is marked as "closed", we will never develop this node again
  - If we have a vertex in open such that its cost is minimal - by adding any positive number to any vertex - the minimality will never change
  - Without the constraint on positive numbers - the above assumption is not true
  - It assumes them to be positive to make the algorithm run faster and does this to avoid considering paths which can't be shorter
- $len(v)$ should be an *estimated distance*, which needs to be updated for every vertex
  - how many times? $\leq n-1$ edges $\Rightarrow n-1$ times should be enough
  - maximum number of edges in a simple path between any two vertices

5.4.2 Bellman-Ford Algorithm

- *Input*: A directed graph G with edge weights $w: E \to \mathbb{R}$ and a source vertex $s \in V$
- *Output*: Either $dist(s, v), \forall v \in V$ or a declaration that G contains a negative cycle

The algorithm is used when the graph might possess negative weights and can even detect negative cycles. If the graph contains one, there is no cheapest path, instead one can make it cheaper by one more walk around said negative cycle (in $n - 1$ iterations it reaches a fix-point, if it doesn't it means a negative cycle exists). Still, it's slower compared to Dijkstra.

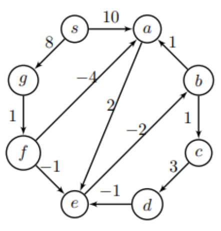
```
procedure $Bellman - Ford\ (G, s)$        ) initial estimated distances
    $len(s) = 0$
    $len(v) = \infty\ \forall v \neq s$
    for $n - 1$ $iterations$ do
        for each $edge\ (u, v) \in E$: do
            $//\ update\ the\ estimated\ distance\ (aka\ “relax”\ edge\ (u, v))$
            $len(v) = \min\{len(v), len(u) + w(u, v)\}$
        for each $edge\ (u, v) \in E$: do
            if $len(v) > len(u) + w(u, v)$ then
                $//some\ distance\ changed\ in\ the\ n - th\ iteration$
                return “$G\ contains\ a\ negative\ cycle$”
```

So basically, it either returns $SSSP(G, s)$ or a declaration G has a negative cycle. It overestimates the length of the path, then slowly goes by relaxation.

Complexity: $O(m * n)$ → a loop working $(n - 1)$ times over m vertices

The following is an example of the algorithm working as code, iteratively reaching multiple vertices and estimating the current distance the best way.

Remember, for each iteration, one needs to use the values of the previous column in order to make the algorithm work properly. Column "last" changes only if the "7" column does:



Iterations

Vertices

	0	1	2	3	4	5	6	7	last
s	0	0	0	0	0	0	0	0	0
a	∞	10	10	5	5	5	5	5	5
b	∞	∞	∞	10	6	5	5	5	5
c	∞	∞	∞	∞	11	7	6	6	6
d	∞	∞	∞	∞	∞	14	10	9	9
e	∞	∞	12	8	7	7	7	7	7
f	∞	∞	9	9	9	9	9	9	9
g	∞	8	8	8	8	8	8	8	8

iterations

vertices

	0	1	2	3	4	5	6	7	last
S	0	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5	5
C	∞	∞	∞	∞	11	7	6	6	6
D	∞	∞	∞	∞	∞	14	10	9	9
E	∞	∞	12	8	7	7	7	7	7
F	∞	∞	9	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8	8

Consider for the exam some examples are present in the relative chapter.

Written by Gabriel R.

Comments:

- It's more "distributed" then Dijkstra ⇒ has played a prominent role in the evaluation of the Internet routing protocols
- It has been the fastest algorithm until 2022, when a near-linear algorithm was published

Correctness:

Let $len(i, v)$ denote the length of a shortest path from s to v that contains $\leq i$ (to read as "at most") edges. Since the shortest path from s to v contains $\leq n - 1$ edges, it's sufficient to prove that after i iterations, $len(v) \leq len(i, v)$ (so, length of the "real" shortest path is $\leq$ the one using at most i edges).

We will prove it by induction on i:

- Base case: $i = 0$

$$len(s) = 0 \leq len(0, s) = 0$$

$$len(v \neq s) = +\infty = len(0, v \neq s)$$

$$//cannot\ reach\ v\ using\ 0\ edges, so\ this\ is\ equal\ to\ the\ path\ not\ using\ s$$

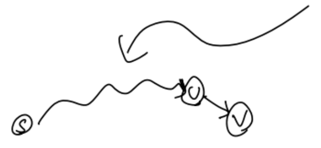
- Inductive hypothesis:

$$len(v) \leq len(k, v)\ \forall 1 \leq k < i$$

Assume this is true for every k and this needs to be proven in the case of at most i edges.

Take $i \geq 1$ and a shortest path from s to v with $\leq i$ edges. Let (u, v) be the last edge of this path. Then:

$$len(i, v) = w(u, v) + \underbrace{len(i - 1, u)}_{\text{why? by contradiction}}$$



Assume taking i edges and taking a shortest path from s to v. We consider (u, v) to be the last edge of this path of the path and the $i - 1$ edge which is the middle edge one in the picture.

We reason by contradiction, arguing there should be a shorter shortest path apart from the one we are considering.

By the inductive hypothesis, $len(u) \leq len(i - 1, u)$. In the $i - th$ iteration we update:

$$len(v) = \min\{len(v), \underbrace{len(u) + w(u, v)}_{\substack{\leq len(i-1,u) + w(u,v) \\ = len(i,v)}}\}$$

In another form, what happens above is the following:

$$len(u) + w(u, v) \leq len(i - 1, u) + w(u, v) = len(i, v)$$

Therefore, we have $len(v) \leq len(i, v)$ as desired.

5.5 ALL-PAIRS SHORTEST PATHS (APSP)

Description:

- *Input*: A directed, weighted graph $G = (V, E)$
- *Output*: One of the following:
    - $dist(u, v)$ ∀ ordered vertex pair
    - A declaration that G contains a negative cycle
        - This can be problematic in finding a shortest path
        - Now we would have to output n^2 shortest paths

A legitimate application of all-pairs shortest paths is to determine the diameter of a network: the longest of all shortest paths.

Obvious solution:

- Invoke Bellman-Ford (B-F) once for every vertex
- It works, but has very high complexity (so, very inefficient): $O(m * n^2)$

Can we do better? Yes, using *dynamic programming*.

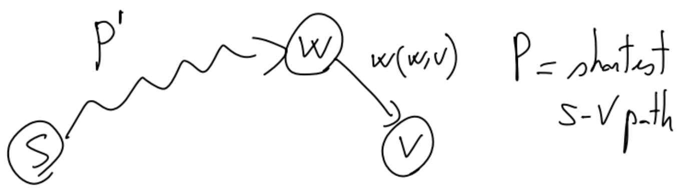
Outline:

- The B-F algorithm has a dynamic programming formulation
- It's not difficult to see one can adapt that formulation to APSP, obtaining an $O(m * n^2)$ algorithm (this will not be seen); an improved formulation can be made to run in $O(n^3 \log(n))$
- A different dynamic programming strategy gives an $O(n^3)$ algorithm (without full proofs)

5.5.1 Bellman-Ford via dynamic programming

What are the subproblems here?

We are gonna see this with an example, considering a shortest $s - v$ path and a subpath P', considered as the prefix of the $s - v$ shortest path



why? by contradiction, as before

Observation: P' is a shortest path (to a ≠ (different) destination, specifically to w) with fewer edges than P. Then P' can be interpreted as a solution to a smaller subproblem.

- *Idea*: introduce a parameter i that restricts the maximum number of edges allowed in a path, with smaller subproblems having smaller edge budgets (measure of subproblem size)

Subproblems: compute $len(i, v)$, the length of a shortest path from s to v that contains at most i edges (if no such path exists, define $len(i, v)$ as $+\infty$) → $O(n^2)$ subproblems.

Observation: every subproblem works with the full input; the idea is to control the allowable size of the output (i.e. solution to a subproblem) – compute a smaller solution on the whole input.

Bellman-Ford recurrence

We will write a recurrence on the costs:

$$len(i,v) = \begin{cases} 0 & i = 0 \text{ and } v = s \\ +\infty, & i = 0 \text{ and } v \neq s \\ min \begin{cases} len(i-1,v) \\ min_{(u,v)\in E}(len(i-1,u) + w(u,v)) \end{cases} & \text{otherwise} \end{cases}$$

(It’s easy to transform a dynamic programming evaluation of this recurrence into our original formulation of B-F). This formulation can be adopted to APSP → $O(n^3 \log(n))$

5.5.2 Floyd-Warshall algorithm

- *Idea*: go one step further; instead of restricting the number of edges allowed in a solution, restrict the identities of the vertices that are allowed in a solution (in other words, now paths can pass through only certain vertices)
    - Basically, it compares many possible paths through the graph between each pair of vertices using intermediate vertices

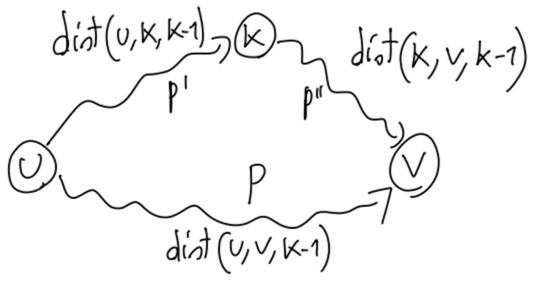
Let’s define the subproblems:

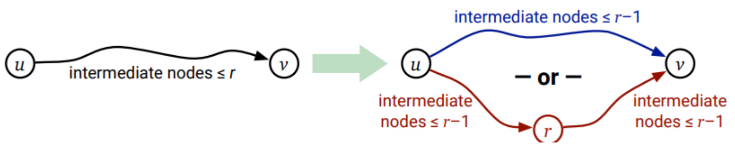
- Call the vertices $1,2,\dots,n$
- Compute $dist(u,v,k)$ = length of a shortest path from u to v that uses only vertices from $\{1,2,\dots,k\}$ as internal (i.e., not u or v) – passes only through them – vertices and that does not contain a directed cycle (if no such path exists, define $dist(u,v,k) = +\infty$)

Consider parameter k defines the sub-problem size. In total, there are $O(n^3)$ sub-problems, considering we have n choices for u, but also n choices for u and n choices for k.

Algorithm: expand the set of allowed internal vertices, one vertex at a time, until this set is V.

Payoff of defining subproblems in this way: there are only 2 candidates for the optimal solution to a subproblem, depending on whether it uses vertex k or not:





So, we consider $\min\{dist(u,v,k-1), dist(u,k,k-1) + dist(k,v,k-1)\}$

Complexity:

- $O(1)$ work per subproblem
- Total complexity: $O(n^3)$

Note that there are only two candidates for the optimal solution to a sub-problem, depending on whether it uses vertex k or not (if path becomes better this way).

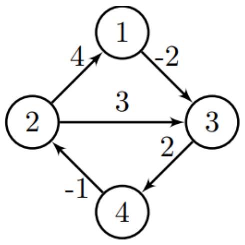
- If the sum of the distance between u and the new node in question k added to the distance between k and v is less than the distance directly between u and v
    - Then the latter is replaced with the previous sum
    - (To go from node u to node v I should go through node k)
- To catch negative cycles with this one, it's enough to check $dist(v, v) \geq 0, \forall v \in V$

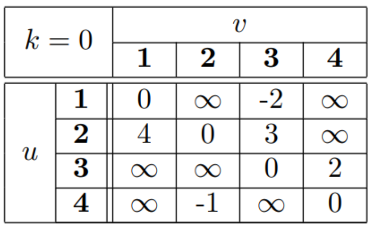
Here goes the algorithm:

```
procedure $Floyd - Warshall(G)$
    $label\ the\ vertices\ V = \{1,2,\dots,n\}\ arbitrarily$
    $//\ subproblems\ (k\ indexed\ from\ 0)$
    $A = n * n * (n+1)\ array\ //\ here\ we\ will\ store\ the\ solutions$
    $//\ base\ cases\ (k = 0)$ – $initialize\ accordingly\ vertices$
    for $u = 1$ to $n$: do
        for $v = 1$ to $n$: do
            if $u = v$ then $A[u, v, 0] = 0$
            else if $(u, v) \in E$ then $A[u, v, 0] = w(u, v)$
            else $A[u, v, 0] = +\infty$
    $//\ solve\ all\ subproblems\ (triple\ loop\ because\ of\ dynamic\ programming)$
    for $k = 1$ to $n$: do
        for $u = 1$ to $n$: do
            for $v = 1$ to $n$: do
                $A[u, v, k] = \min\{A[u, v, k-1], A[u, k, k-1] + A[k, v, k-1]\}$
    for $u = 1$ to $n$: do $//\ if\ diagonal\ is\ different\ from\ 0, there\ are\ no\ negative\ loops$
        if $A[u, u, n] < 0$ then return "$G\ contains\ a\ negative\ cycle$"
```

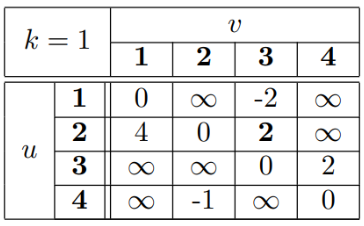
Is there a truly-subcubic algorithm for APSP? $O(n^{3-\epsilon})$ for some constant $\epsilon > 0$. This one is still an open problem in Computer Science to solve.

Consider this execution example (added by me, not seen in class):

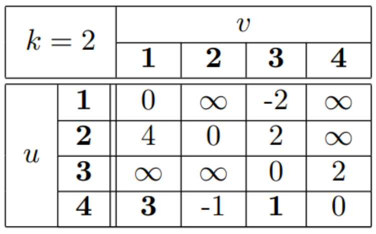




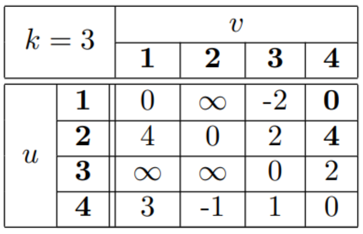
$k = 0$		v			
		1	**2**	**3**	**4**
u	**1**	0	∞	-2	∞
	2	4	0	3	∞
	3	∞	∞	0	2
	4	∞	-1	∞	0



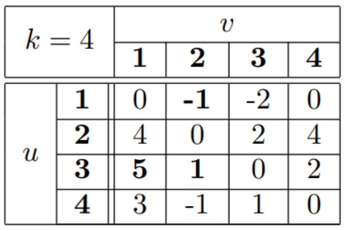
$k = 1$		v			
		1	**2**	**3**	**4**
u	**1**	0	∞	-2	∞
	2	4	0	**2**	∞
	3	∞	∞	0	2
	4	∞	-1	∞	0



$k = 2$		v			
		1	**2**	**3**	**4**
u	**1**	0	∞	-2	∞
	2	4	0	2	∞
	3	∞	∞	0	2
	4	**3**	-1	**1**	0

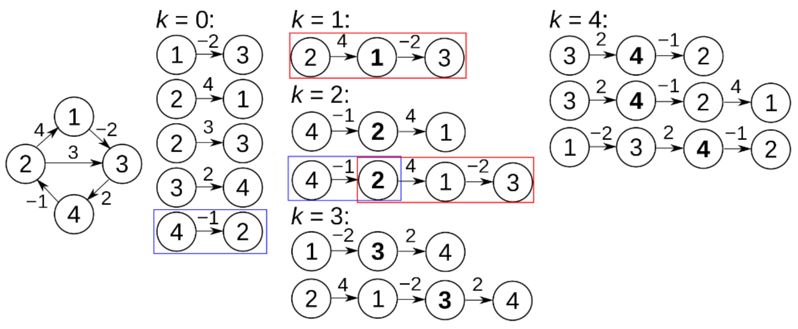


$k = 3$		v			
		1	**2**	**3**	**4**
u	**1**	0	∞	-2	**0**
	2	4	0	2	**4**
	3	∞	∞	0	2
	4	3	-1	1	0



$k = 4$		v			
		1	**2**	**3**	**4**
u	**1**	0	**-1**	-2	0
	2	4	0	2	4
	3	**5**	**1**	0	2
	4	3	-1	1	0

To be precise (taken from Wikipedia this one, but clear enough):

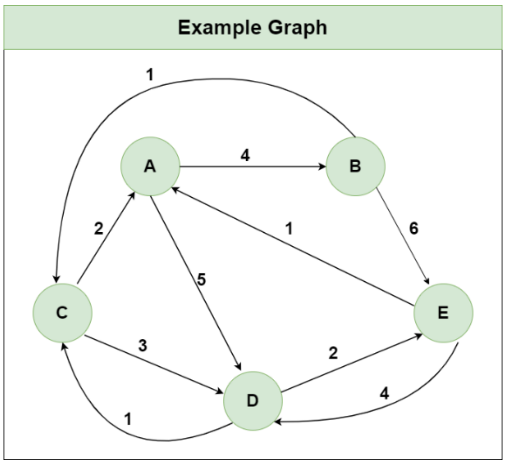


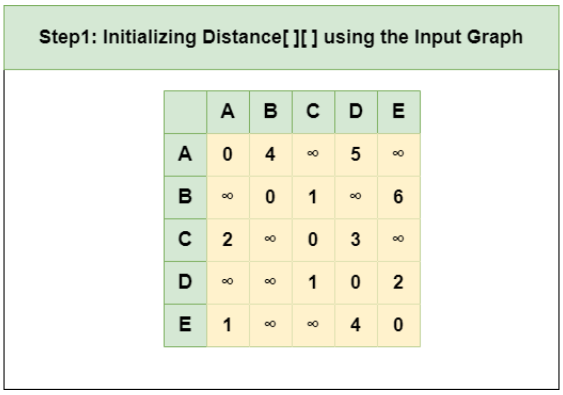
See the algorithm at work here.

Demonstration of Floyd-Warshall algorithm for all-pairs shortest path on a directed graph with 4 vertices.

- At $k = 0$, prior to the first iteration of the outer loop, the only known paths correspond to single edges in the original graph. Values based on edges, zero-encoding the diagonal so to know exactly how many edges there are-.
- At $k = 1$, paths that go through the vertex 1 are found: in particular, the path $2 \to 1 \to 3$ is found, replacing the path $2 \to 3$ which has less edges but is longer. Starts with the loop
- At $k = 2$, paths going through the vertices $\{1,2\}$ are found. The red and blue boxes show how the path $4 \to 2 \to 1 \to 3$ is assembled from the known paths $4 \to 2$ and $2 \to 1 \to 3$ encountered in previous iterations. The path $4 \to 2 \to 3$ is not considered, because it is already known that $2 \to 1 \to 3$ is the shortest path from 2 to 3
- At $k = 3$, paths going through the vertices $\{1,2,3\}$ are found
- Finally, at $k = 4$, all shortest paths are found

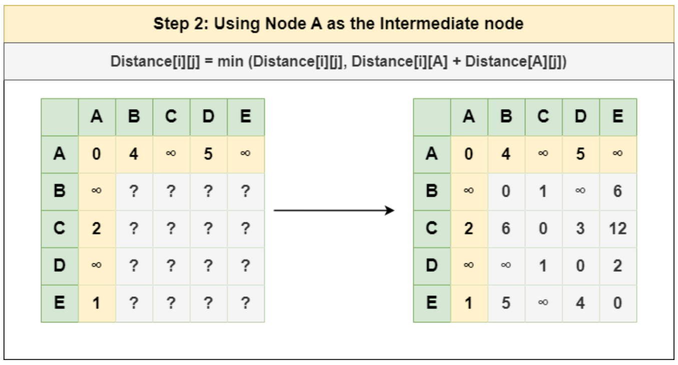
Another two examples:





Step1: Initializing Distance[ ][ ] using the Input Graph

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0



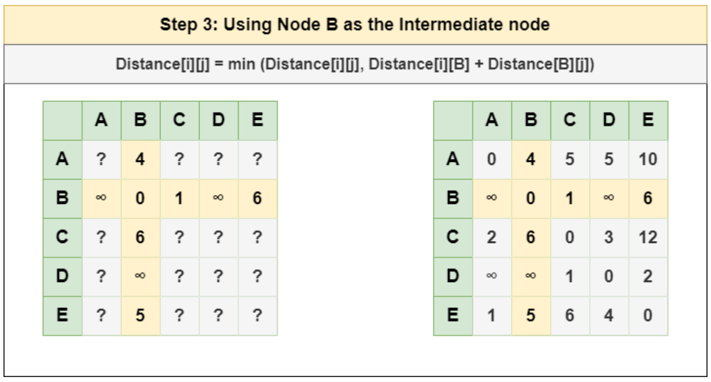
Step 2: Using Node A as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

$\longrightarrow$

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

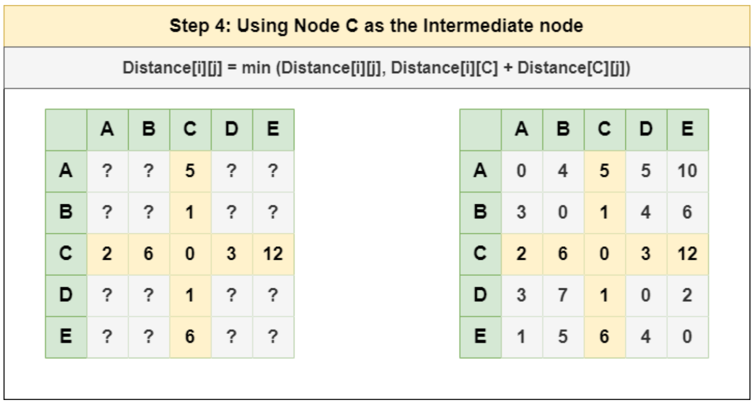


Step 3: Using Node B as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

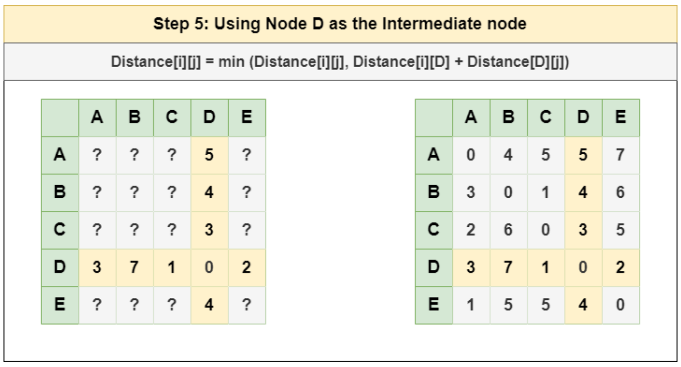


Step 4: Using Node C as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

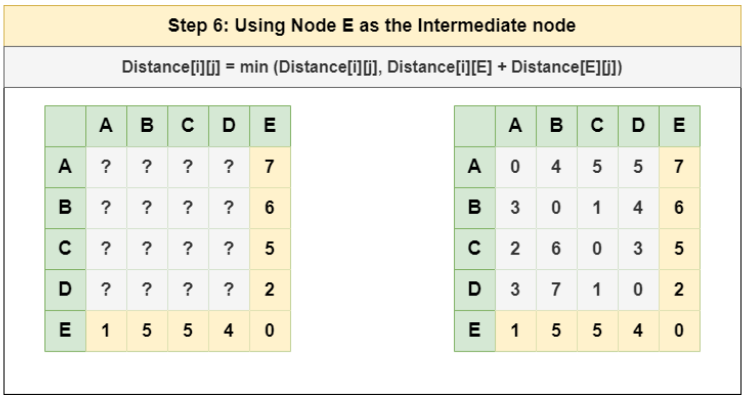
	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0


Step 5: Using Node D as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

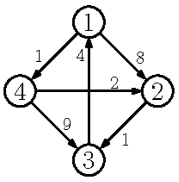
	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0


Step 6: Using Node E as the Intermediate node

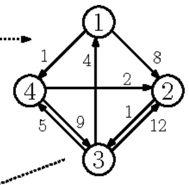
Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

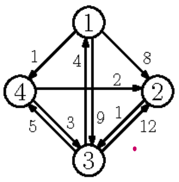
	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0



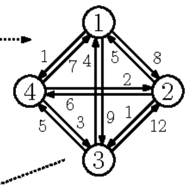
$$d^{(0)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



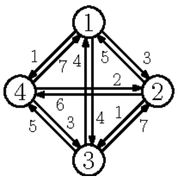
$$d^{(1)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



$$d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

$$\text{final} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

6 Maximum Flows

(This is the last part of the course about graphs and in this period, the professor will upload a past exam and other exercises. Further readings: survey, paper, article)

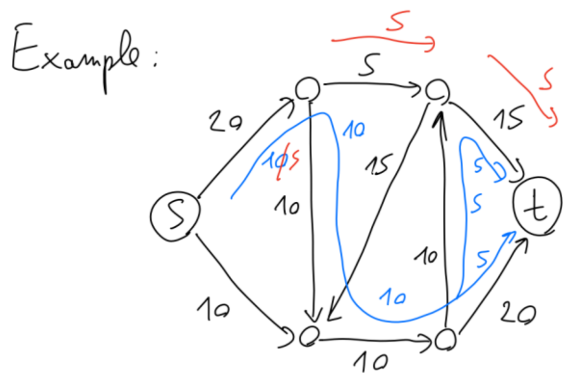
This problem originated in the 50's to study rail networks. We give a couple of definitions:

- Flow network is a directed graph $G = (V, E)$ where each edge has a capacity $c(e) \in \mathbb{R}^+$, along with a designated source $s \in V$ and sink $t \in V$:
    - For convenience, write $c(e) = 0$ if $e \notin E$, no edges enter s and no edges leave t
- Flow is a function $f: E \to \mathbb{R}^+$ satisfying the following constraints:
    - *Capacity*: $\forall e \in E, f(e) \leq c(e)$ – value of the flow at most capacity of that edge
    - *Conservation*: $\forall u \in V \setminus \{s, t\}$ we have
    $$\sum_{u \in V \text{ s.t.} (v,u) \in E} f(v, u) = \sum_{v \in V \text{ s.t.} (u,v) \in E} f(u, v)$$
    - *Conservation of flows*: the amount of flow going in nodes must be equal to the flow going out from those
        - Initially, such flow is 0, which is "how much we can pass on the edge"
- Value of a flow is:
$$|f| = \sum_{v \in V \text{ s.t.} (s,v) \in E} f(s, v)$$
    - Basically, the sum of all flows going in and out vertices thanks to edges
    - As a matter of fact, the amount of stuff traveling from source to sink
    - Such flow has to be less than or equal to the capacity

6.1 Maximum Flow Problem

The maximum flow problem can be described as follows: given a flow network, find a flow f of maximum value. Such flow is measured on *the maximum value received in a sink node*.

Consider the following example of a network going from source to sink, each edge weighted representing capacities. We want to always retain the minimum, called bottleneck – see later.



What was identified was a flow of value 10. In blue, the flow, which is the minimum capacity going forward, allowing to not exceed edges. First is 10, then can become 5 or only 5 (red one).

There can be several *applications*:

- Rail/Airline/Road networks
- Electrical networks
- Liquid transportation networks
- Moreover, it can be applied to solve several other problems in Computer Science
  - (e.g., bipartite matching)

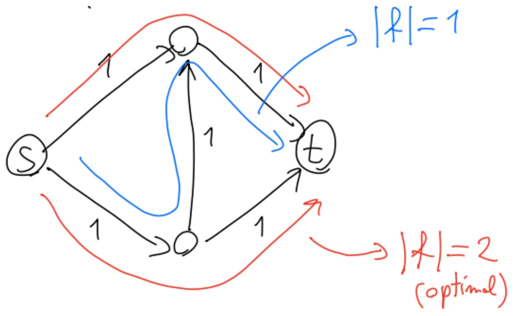
Max flow reduces to linear programming (like many other problems) but there are more efficient special-purpose algorithms. We'll see one of them (Ford-Fulkerson), but there are plenty of more efficient algorithms – see "Further reading" section of Moodle or this chapter beginning link.

A natural idea as first simple algorithm – solve the problem in a greedy way:

- Find a path from s to t (in linear time using BFS)
- Send as much flow along it as possible
- Update capacities
- Remove edges that have 0 remaining capacity
- Repeat until the graph has no $s - t$ graphs

Note: this will <u>not</u> always work, however.

Consider the following counterexample: here we have all edges with same capacity and choosing on path instead of one another can make a difference. The greedy algorithm fails in this example because it only considers one path at a time and does not account for the possibility of finding multiple paths that, when combined, can yield a higher total flow.



Idea to improve this algorithm: revise/undo some of this flow <u>later</u> in the algorithm. How? By "pushing back" some flow through new edges in the reverse direction.

6.2 Ford-Fulkerson Algorithm

Definition: given a flow network G a flow f, the <u>residual network</u> of G w.r.t (with respect to) flow f, G_f, is a network with vertex set V and with edge set E_r as follows:

- For every edge $e = (u, v)$ in G
  - If $f(e) < c(e)$, add e to G_f with capacity $C_f(e) = c(e) - f(e)$
  - If $f(e) > 0$, add another edge (v, u) to G_f with capacity $C_f(e) = f(e)$

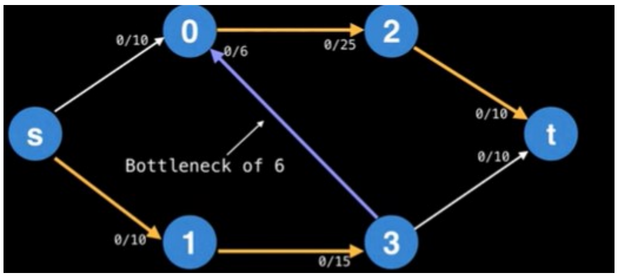
You might be surprised that the residual network G_f can also contain edges that are not in G. As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge in order to increase the flow elsewhere.

Written by Gabriel R.

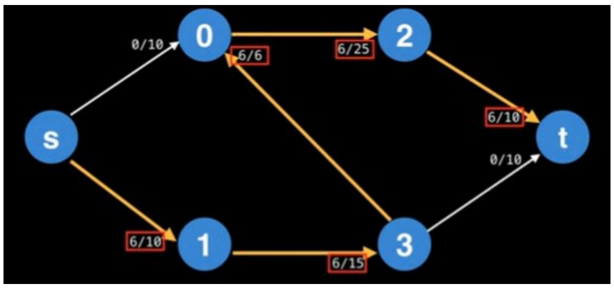
The Ford-Fulkerson (F-F) algorithm (1956) repeatedly finds an $s - t$ path P in G_f (e.g., using BFS) and uses P to increase the current flow.

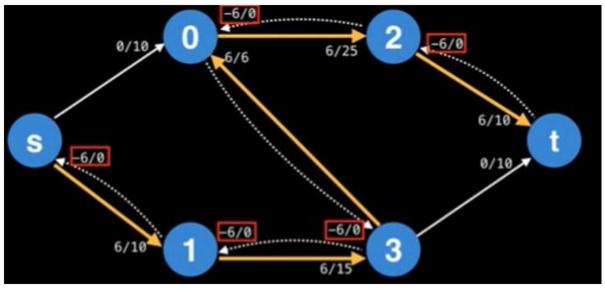
- P is called augmenting path
    - This is a path of edges in the residual graph with unused capacity greater than 0 from the source s to the sink t
    - This can only flow on edges not fully saturated yet
- In an augmenting path, the *bottleneck* is the smallest edge on the path
    - We can use this one to augment the flow along the path

In figure below, in orange the augmenting path, in light-blue as written the bottleneck:



- Augmenting the flow means updating the flow values along the augmenting path (left)
    - For forward edges, this means increasing the flow by the bottleneck value
- When augmenting the flow along the augmenting path
    - You also need to decrease the flow along each residual edge (backward edges) by the bottleneck value (right)
    - Residual edges exist to "undo" bad augmenting paths which do not lead to a maximum flow





The residual graph, so, contains also residual edges. This algorithm continues to find augmenting paths and augments the flow until no more augmenting paths exist.

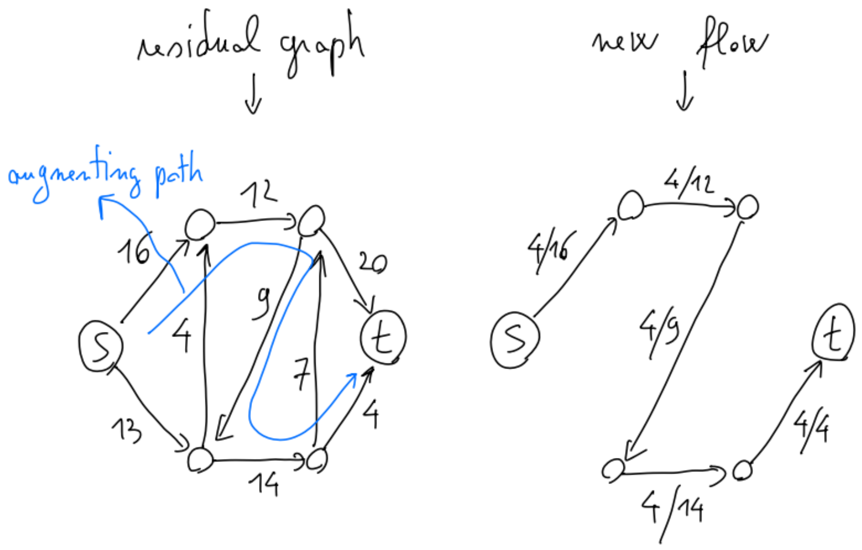
- A key realization is that the sum of the bottlenecks found in each augmenting path is equal to the max flow
- Each time you take the minimum capacity value, reducing all capabilities of the path and increasing back edges capacities

Note: there is no criteria in the path selection. Infact, is selects an s-t path, but "brute-forces" each one so to have no more augmenting paths in the end.

Let’s dive into the pseudocode.

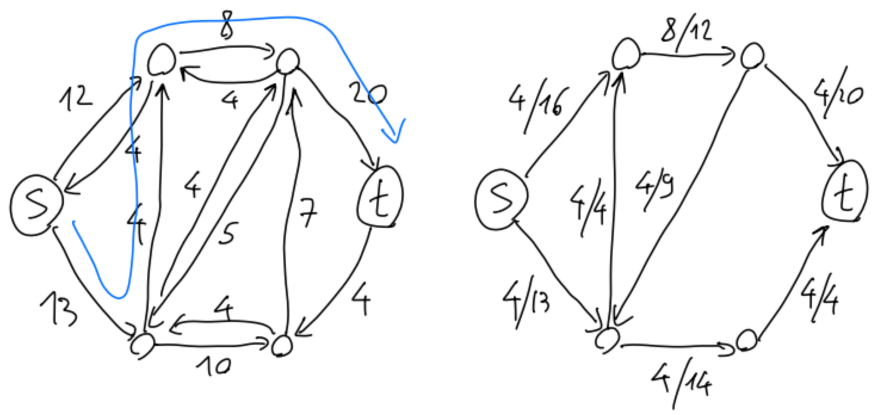
```
procedure $Ford - Fulkerson\ (G, s, t)$
    $initialize\ f(e) = 0$ for all $e \in G, E$
    $G_f = G$
    while $there\ exists\ an\ augmenting\ path\ P\ in\ G_f$: do
        $let\ \Delta_p = \min_{e \in P} C_f(e)$ $//\ \Delta_p\ is\ the\ \text{“}bottleneck\text{”}\ capacity\ in\ P$
        for each $edge\ e = (u, v) \in P$: do
            if $(u, v) \in G, E$ then:
                $f(u, v) = f(u, v) + \Delta_p$ $//\ send\ flow\ along\ the\ path$
            else $f(v, u) = f(v, u) - \Delta_p$ $//\ the\ flow\ might\ be\ \text{“}returned\text{”}\ later$
        $update\ the\ residual\ graph\ G_f$
    return $f$
```

Consider the following example, first with residual graph (edges each with its own capacity), then we consider the algorithm being applied. Basically, here we consider 4 which is the “bottleneck” – ergo, the minimal capacity. We are considering *one generic* $s - t$ path, then trying all the others.

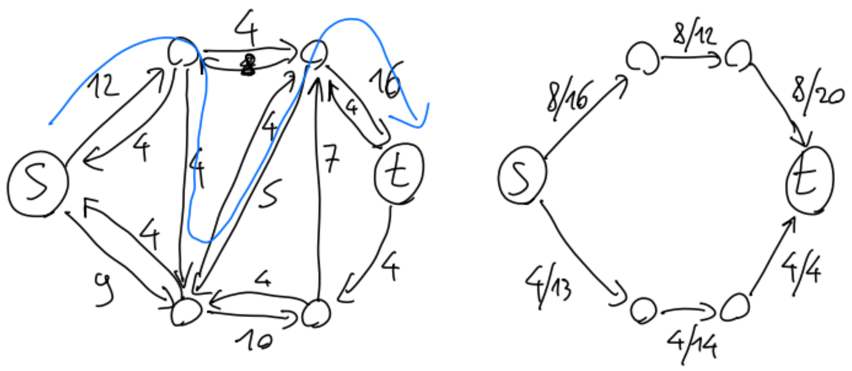


The input graph here will not be modified, given we are working with the residual graph, so capacities get updated accordingly. To allow for “reviewing” the past decisions, we add edges going back, simply marking the fact we chose a capacity that allows us to “balance how much we have as of now”.

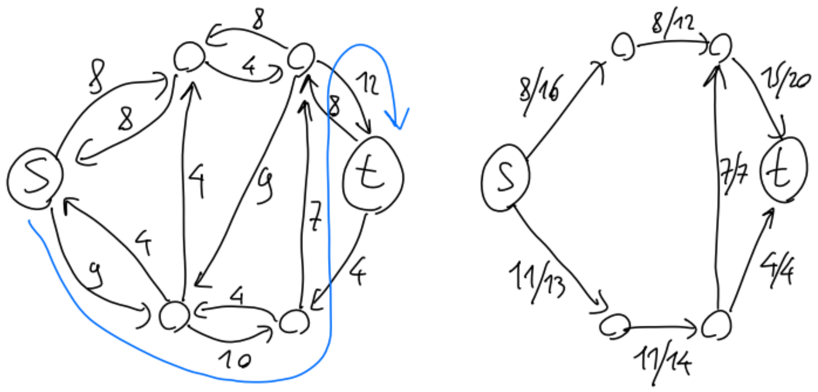
Edits are made according to how many units more we are sending, with respect to the capacity of single nodes. Again, a generic $s-t$ path is chosen, then the minimum capacity (4) gets chosen for all nodes and the according residual capacity gets updated on the graph.



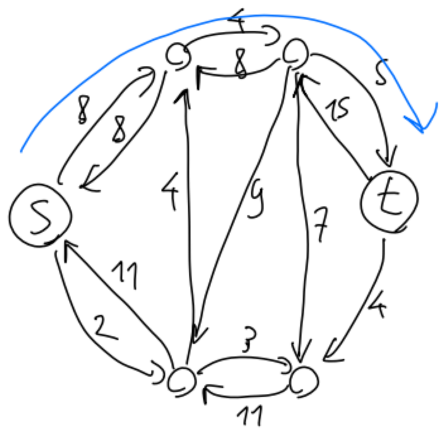
Now the capacity getting selected is 4, so we update each edge of residual network. This $s-t$ path is pushing back some flow, revising its decision from the original one. As you can see, one edge which had 4 before now goes back given it has spent its full capacity and another one which reached its limit (9), being "eliminated" from the graph, given it goes back from its decisions.

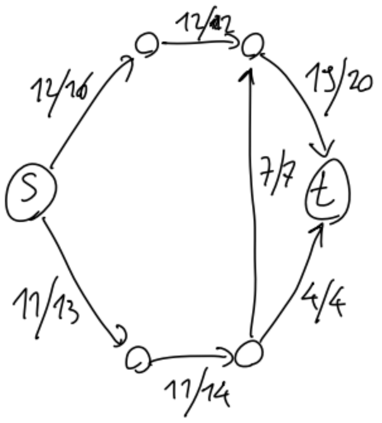


Again, minimum capacity is 7, so in the new flow we consider each path.

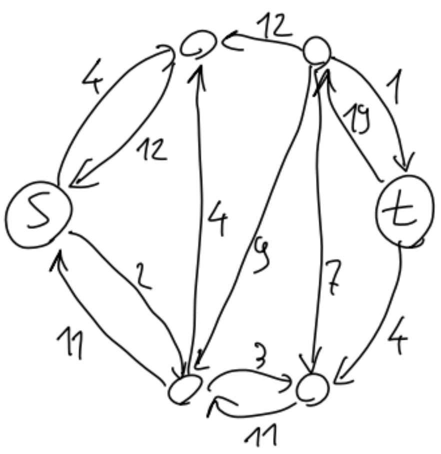


One more iteration to go, using the same principle:





The algorithm stops where there are no more $s - t$ paths, thing that happens here:



There are no more augmenting paths, and this implies this one is a max flow of value 23.

Basically, when we find an augmenting path P:

- if an edge $(u, v) \in E$, we are increasing its flow by the bottleneck capacity
- if an edge $(u, v) \notin E$, it's like we are decreasing the flow on the edge (v, u) and adding that amount of flow to other edges

Complexity:

- Assume capacities are integers; then:
  - the flow value increases by ≥ 1 is each iteration
  - the complexity of each iteration is $O(m)$
- Total complexity is $O(m * |f^*|)$, where f^* is a max flow

To fully summarize and complete (since the same example comes from CLRS):

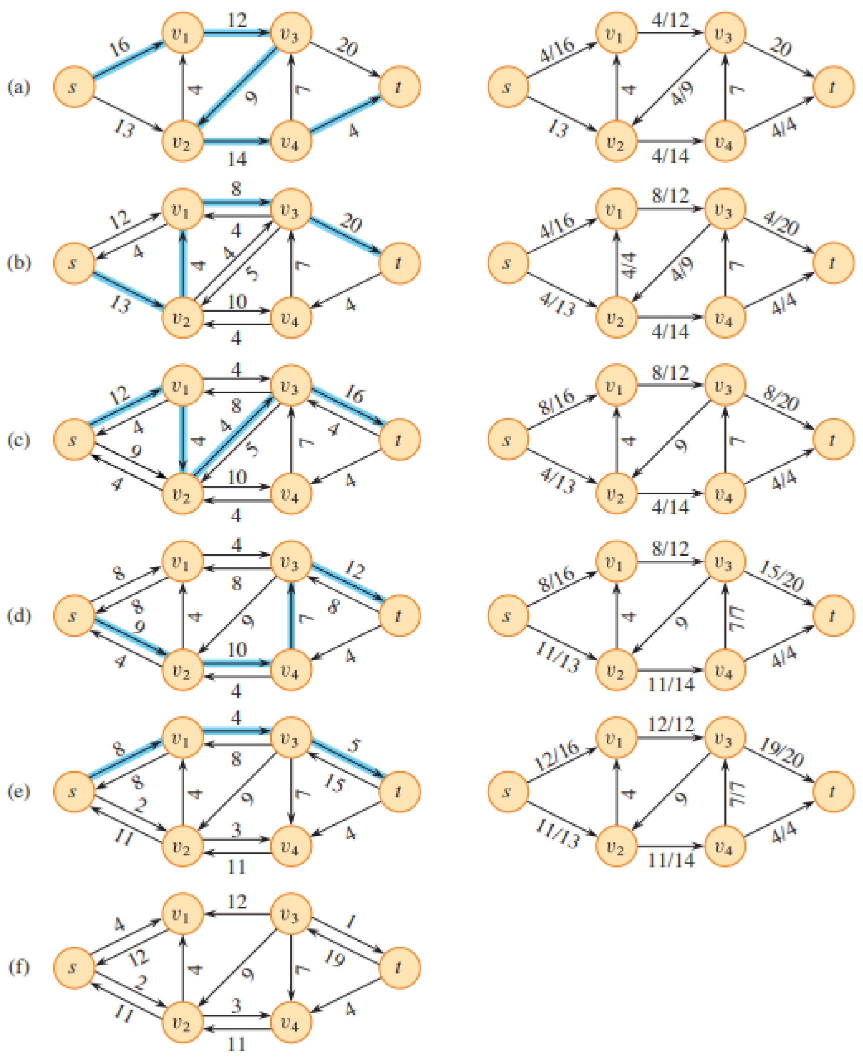
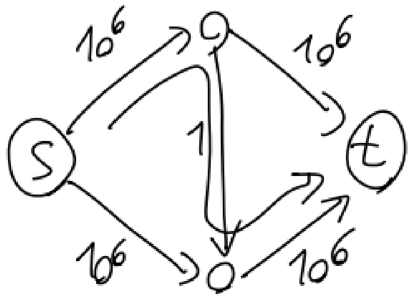


Figure 24.6 The execution of the basic Ford-Fulkerson algorithm. **(a)–(e)** Successive iterations of the **while** loop. The left side of each part shows the residual network G_f from line 3 with a blue augmenting path p. The right side of each part shows the new flow f that results from augmenting f by f_p. The residual network in (a) is the input flow network G. **(f)** The residual network at the last **while** loop test. It has no augmenting paths, and the flow f shown in (e) is therefore a maximum flow. The value of the maximum flow found is 23.

In case, even the gif of the algorithm executing.

A flow network for which F-F can take $\theta(m * |f^*|)$ time:



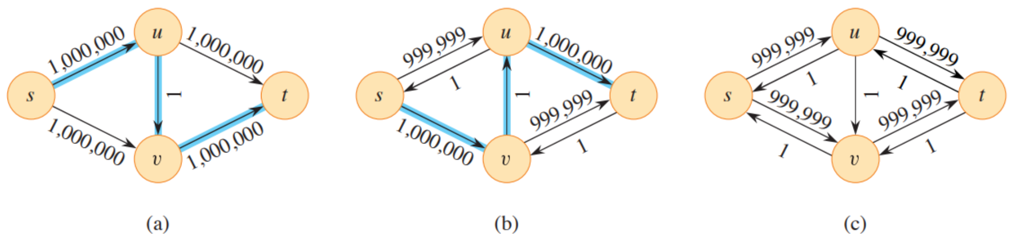
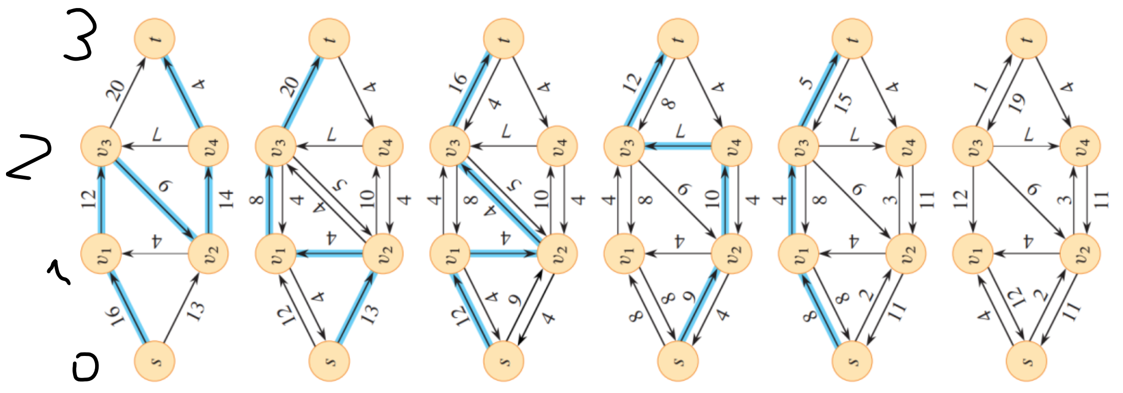


Figure 24.7 **(a)** A flow network for which FORD-FULKERSON can take $\Theta(E\,|f^*|)$ time, where f^* is a maximum flow, shown here with $|f^*| = 2{,}000{,}000$. The blue path is an augmenting path with residual capacity 1. **(b)** The resulting residual network, with another augmenting path whose residual capacity is 1. **(c)** The resulting residual network.

Input size: $O(mlog(U))$, where U = max capacity.

In total for the complexity: $O(m|f^*|) = O(m * n * U)$ "pseudo-polynomial".

Note: it uses BFS (goes in levels – see drawing of mine using CLRS example) and tries all paths so to consume the flow.



7 2ND PART OF THE COURSE - NP-HARDNESS

(Further readings and material: article, paper, paper, article, paper and video)

Preamble: (a primer on NP-Hardness)

- In the 30s, we started to understand what is or isn't effectively computable
  - E.g., Halting Problem, generally problems which cannot be solved by computers
- By the 60s, computer scientists had developed fast algorithms to solve some problems
  - While for others the only known algorithms were very slow
- In the 70s, we started to understand what is or isn't efficiently computable
  - This so-called $P \neq NP$ question has been one of the deepest, most perplexing open research problems in theoretical computer science, since it was first posed in 1971

In 1965, Jack Edmonds defined what efficient means: an algorithm is "efficient" if its running time is $O(n^k)$ for some constant k ($n =$ input size) (for the curious of you, base of Cobham-Edmonds thesis, which basically specifies the following):

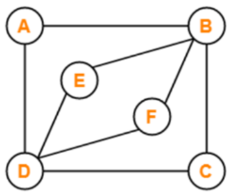
- Problems for which a polynomial time algorithm exists are called tractable
  - all the algorithms seen so far
- If no polynomial time algorithm exists then the problem is called intractable

7.1 PROBLEMS

Examples to illustrate how perplexing questions about efficient computation can be:

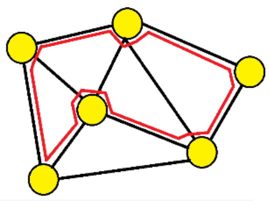
almost the same

1) *Eulerian Circuit problem*
   a. Given an undirected graph, an Eulerian Circuit is a cycle that traverses *all the edges only once*
   b. This problem can be solved in linear time (exercise)
   c. Note that in an Euler circuit you *might* pass through a *vertex more than once*



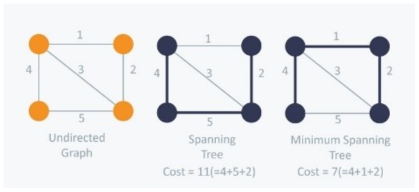
Euler Circuit = ABCDFBEDA

2) *Hamiltonian Circuit problem*
   a. Given an undirected graph, an Hamiltonian Circuit is a cycle that traverses all the *vertices* only once (NOT every edge)
   b. To date, no one knows a polynomial algorithm to solve it!
   c. Note that in a Hamiltonian circuit you *may not pass* through all *edges* (see right figure)



almost the same

3) *Minimum Spanning Tree*
   a. Given a connected, undirected graph and a function $w: E \to \mathbb{R}$, output a spanning tree $T \subseteq E$ minimizing $\sum_{e \in T} w(e)$



4) *Traveling Salesperson Problem (TSP) – before was Salesman*
   a. Given a complete, undirected graph and a function $w: E \to \mathbb{R}$, output a *tour* $T \subseteq E$ (i.e. a cycle that visits every vertex exactly once, basically an Hamiltonian circuit) minimizing $\sum_{e \in T} w(e)$
   b. To date, no one knows a polynomial algorithm to solve it!

A much easier task: given a graph and a list of vertices C, check (as opposed to output) if C is an Hamiltonian circuit.

- Problems that are *easy* to *solve*: class P ("polynomial time")
  - (1) and (3) $\in P$
- Problems that are *easy* to *verify*: class NP ("nondeterministic" polynomial)
  - (1), (2), (3), (4) $\in NP$
  - Rookie mistake: $NP \neq$ not-polynomial

7.1.1 Exercises

Problem:

Given an undirected graph, an eulerian circuit is a cycle that traverses all the edges only once.

Show it can be solved in linear time.

Solution (not an official one, but still, I think it's sound enough considering it's based on research)

To solve the problem of finding an Eulerian circuit in an undirected graph in linear time, we can use the following algorithm:

1. Check if the graph is connected and has at most two vertices with odd degrees. If there are more than two vertices with odd degrees, then an Eulerian circuit cannot exist.
2. If there are exactly two vertices with odd degrees, start the Eulerian circuit at one of them. Otherwise, start from any vertex.
3. Traverse the graph using the following strategy:
   - At each vertex, choose an unvisited edge (if one exists) and traverse it.
   - If there are no unvisited edges at the current vertex, backtrack to the previous vertex.
4. If you can traverse all the edges and end up at the starting vertex, then an Eulerian circuit exists. Otherwise, an Eulerian circuit does not exist.

This algorithm works in linear time because it visits each edge exactly twice (once during the traversal and once during backtracking) and performs constant-time operations at each vertex. Therefore, the time complexity is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

There is an existing algorithm doing this done by *Hierholzer*, which showed the sufficient condition in Euler theorem, which states "a graph is connected if an only if has all nodes of even degree or if it has exactly two nodes of even degree". It works for both directed and undirected graphs and works as follows:

Preconditions:

- All vertices in the graph must have even degrees

Steps:

- Start at any vertex (each one can be a starting point) and follow a trail of edges until returning to the starting vertex. This forms a partial circuit

- If the partial circuit covers all edges, the algorithm is complete. Otherwise, select any vertex in the current circuit that has unused edges and start a new circuit from that vertex, merging it into the previous circuit

- Repeat step 2 until all edges have been used
    a. At some point, we will visit a vertex and there will be no edges to follow
    b. Remember that Eulerian Cycle properties, every vertex should have even degrees or equal in-out degrees
    c. If we are stuck the first time it means that we formed a cycle, and the vertex that we are stuck on is the starting vertex. This means we returned where we started.

- The algorithm terminates when a complete Euler circuit is formed, where each edge is traversed exactly once, backtracking from the whole stack and holding complete knowledge of the structure

Hierholzer's algorithm has a linear runtime, making it an efficient method for finding an Euler circuit in a graph that meets the necessary requirements.

7.2 NP-HARD

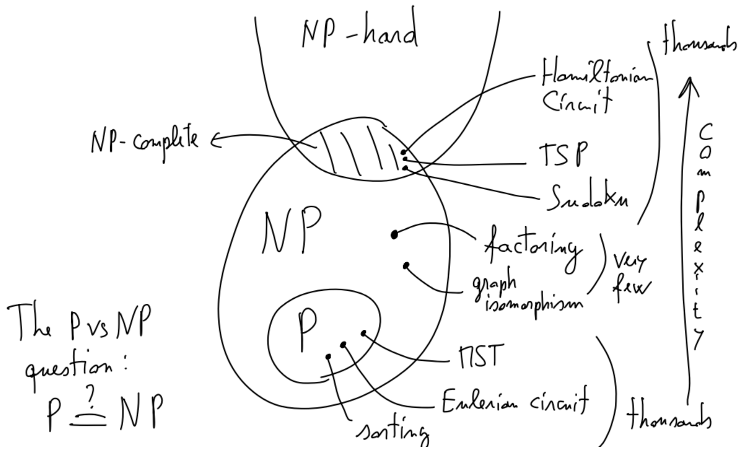
To simplify the study of the complexity of problems (branch of CS called "computational complexity"), we limit out attention to the following class of problems: decision problems. These are problems with a Boolean answer $= \begin{cases} YES \\ NO \end{cases}$. We define the following *complexity classes*:

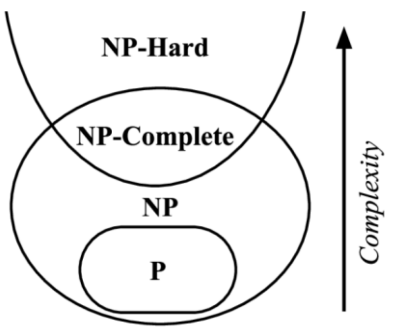
1) P is the set of decision problems that can be solved in polynomial time
2) NP is the set of decision problems with the following property:
    a. If the answer is YES, then there is a proof of this fact (called "certificate") that can be checked in polynomial time
3) $co - NP$, which is essentially the opposite of NP:
    b. *Property*: if the answer is NO, then there is a proof of this fact that can be checked in polynomial time

Now we introduce the concept of *NP-hardness*:

- A computational problem is NP-hard if a polynomial-time algorithm for it would imply a polynomial-time algorithm for every problem in NP
- A problem is NP-Complete (NPC) if it's both in NP and $NP - Hard$
    a. Specifically, this is the complexity class representing the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time
    b. Intuitively, this means we can solve Y quickly if we know how to solve X quickly
- NP-Hard are the problems that are at least as hard as the NP-complete problems and are known to be the hardest in NP

Let's draw a picture of what (today) we think the world looks like:





One of the main open questions in Computer Science is whether $P = NP$.

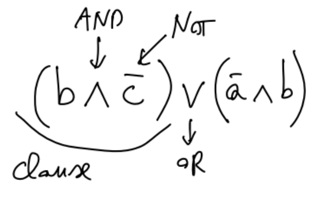
Why study NP-hardness:

- Being NP-hard is strong evidence that a problem is intractable
- It suggests you should use a different approach, such as:
    a. Identify tractable special cases
    b. Compromise on correctness, finding an approximate solution
        i. Approximation algorithms
    c. Use randomness to get a solution correct with high probability
- ... instead of looking for an efficient algorithm which may not even exist!

The point of study the NP-hardness is that knowing that a problem is "intractable" is the best way you have to save some time while working on it, typically focusing on identify tractable special cases of the problem or producing an approximation algorithm.

7.3 COOK-LEVIN THEOREM

The Cook–Levin theorem (1971, made by both scientists at the same time) states that the Boolean satisfiability problem is NP-complete.



- SAT: formula satisfiability (also called B-SAT/SATISFIABILITY)
    a. *Input*: a Boolean formula like the one you see on the right
    b. *Output*: it is possible to assign Boolean values to the variables a, b, c ... so that the entire formula evaluates to TRUE?

We consider a special case of SAT:

- 3-SAT (aka 3-CNF-SAT): a Boolean formula in *conjunctive normal form* (CNF) if it is a *conjunction* (AND) of several *clauses* (so, what's inside the parentheses) each of which is the *disjunction* (OR) of several literals, each of which is either a variable of its negation
    a. Example: $(a \lor \mathrm{b} \lor \overline{c}) \land (b \lor \neg c \lor \neg d) \land (\neg a \lor c \lor d)$
- Basically, a 3-CNF formula is a CNF formula with exactly 3 literals per clause

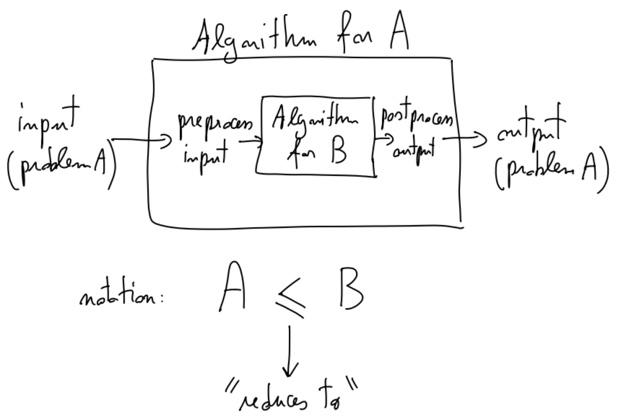
How can we show that a problem is NP-Hard?

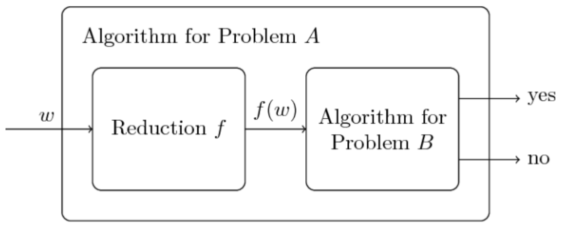
7.4 Reductions (Schema and Formal Definition)

Here we give a general scheme/concept of this one.

To prove that a problem is NP-hard, we use a reduction, which is an algorithm for transforming one problem into one another. It is the way we compare the computational complexity of two problems, A and B.

Generally, a problem A reduces to problem B if an algorithm that solves B can be translated into one that solves A. A classic way to see this is the following one:



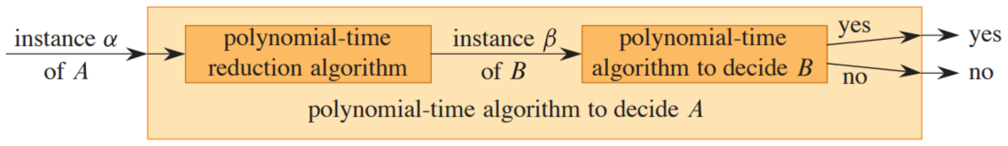


If the reduction is "efficient" then B is as hard as A (equivalently, A is not harder than B).

Definition: A problem A (pre/post processing has to take at most polynomial time – has to be efficient) reduces in polynomial time to problem B ($A \leq_p B$) if there exist a polynomial time algorithm that transforms an arbitrary input instance a of A into an input instance b of B such that:

1) a is a YES instance of $A \Rightarrow b$ is a YES instance of B
2) b is a YES instance of $B \Rightarrow a$ is a YES instance of A

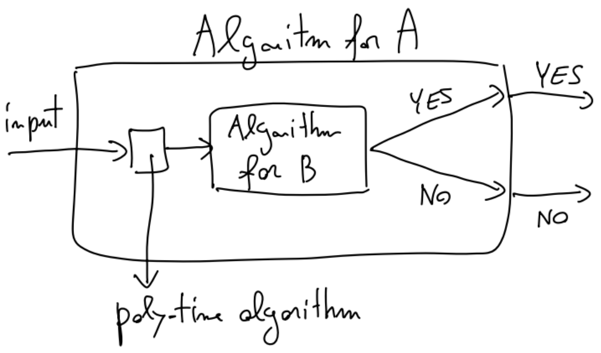
usually called "Karp reduction"


instance α of A
polynomial-time reduction algorithm
instance β of B
polynomial-time algorithm to decide B
yes
no
yes
no
polynomial-time algorithm to decide A

Note: probably you saw the "mapping reduction" → $\leq_m$

Here, we will use the polynomial reduction or, as already mentioned, Karp reduction → $\leq_p$

(For some reason, in past years was only $<_p$)



Written by Gabriel R.

Observation:

- It's more restrictive than the general scheme
  a. only one call to B, no postprocessing needed, only deal with decision problems

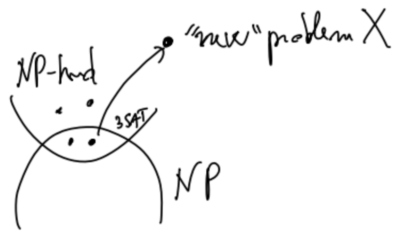
Property (transitiveness): $A \leq_p B$ and $B \leq_p C \Rightarrow A \leq_p C$

7.5 NP-HARDNESS (FORMAL DEFINITION AND PROOF)

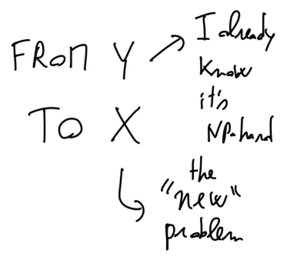
Definition: A problem is NP-Hard if every problem in NP reduces in polynomial time to it.

Then, to prove that a problem X is NP-Hard reduce a known NP-Hard problem Y to X.

- This in turn means that you start from a problem already known to be hard
  a. e.g. 3-SAT
- To *your* problem
  a. E.g. when you say you use 3-SAT for your problem, it implies that you are showing that if you could solve your problem efficiently, then you could also solve 3-SAT efficiently.
  b. Since 3-SAT is NP-complete, any problem that can be reduced to it in polynomial time is NP-hard. Let's *emphasize* this one:



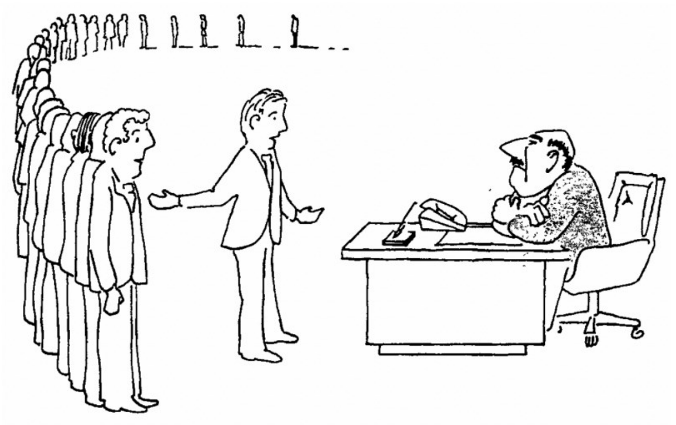
- The reduction is FROM Y
  a. I already know it's NP-hard (known problem)
- to X
  a. the "new" problem



So, do it like $Y \leq_p X$.

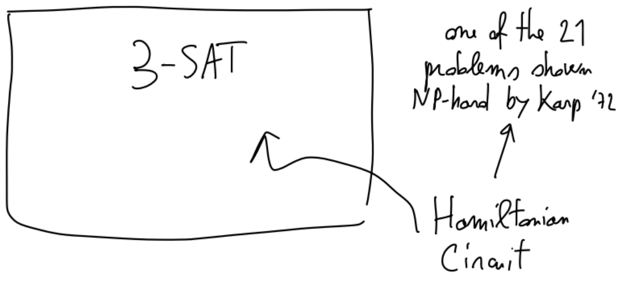
Rookie mistake: do a reduction in the wrong direction (so DO NOT DO $X \leq_p Y$)

Again, NP-hardness doesn't mean the problem is not in P, but it does provide strong evidence for that (so, the problem *may not* be in P) – below, a good cartoon with reference here.



"I can't find an efficient algorithm, but neither can all these famous people."
(picture from the book of Garey & Johnson, 1979)

There is a whole library of NP-Hard problems (complete list here):



Our first NP-hardness problem:

- *Theorem*: TSP (Traveling Salesperson Problem) is NP-Hard
- *Proof*: Reduction from Hamiltonian circuit to TSP ($Ham \leq_p TSP$)

Wait a minute: TSP is not a decision problem! No worries. Define TSP as:

- *Input*: $G = (V, E)$ complete, undirected, weighted graph $k \in \mathbb{R}$
- Output: $\exists$ in G a Hamiltonian circuit of cost $\leq k$ (at most k)?

We could try to use all possible k values, but k is not guaranteed to be polynomial; using only the cycles will not work either.

What we actually do: Pick an arbitrary input instance for $Ham.$ and create the following input for TSP:

- $G' = (V, E')$ complete, undirected, weighted graph with:

$$w(e \in E') = \begin{cases} 1, & if\ e \in E \\ +\infty, & otherwise \end{cases}$$

If we use $k = n$ this reduction takes poly-time (($O(n^2)$). Then:

- if G has an Hamiltonian circuit, then the TSP algorithm run on G' returns an Hamiltonian circuit with cost n
- if G doesn't have a Hamiltonian circuit, then any Hamiltonian circuit in G' must have ≥ 1 edge not in G, hence of weight ∞. Hence, in this case, a TSP algorithm run on G' returns a Hamiltonian circuit of cost $> n$

If we had a fast algorithm for TSP we would also solve the Hamiltonian circuit problem.

7.6 Maximum Independent Set

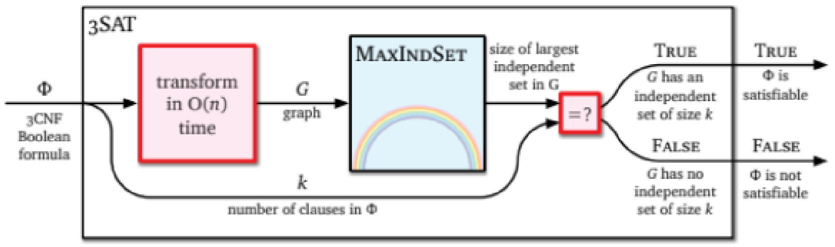
There are more problems we can represent here:

- Independent Set
    a. given a graph $G = (V, E)$ an *independent set* in G is a subset $I \subseteq V$ with no edges between them
- (Maximum) Independent Set (it's considered trivial that the problem is to find the maximum one, so that's why you find braces here – from now on, it will be only "Independent Set")
    a. compute an independent set of maximum size

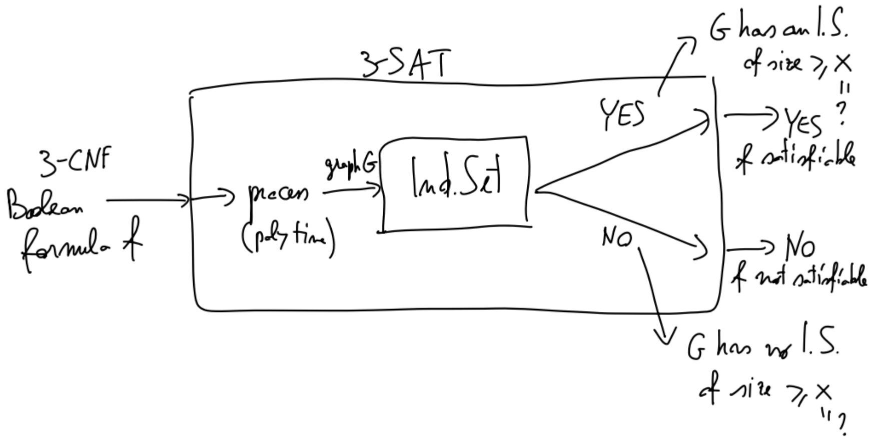
Theorem: Independent Set is NP-Hard

Proof: Reduction from $3SAT$ (problem in logic) to $Independent\ Set$ (problem in graphs) → $3SAT \leq_p IndependentSet$

They seem totally unrelated problems, but let's see what we have to do (figure here is from "Algorithms" book of Jeff Erickson, suggested in particular for the whole NP-Hardness chapter):



What we are conjecturing is the following:



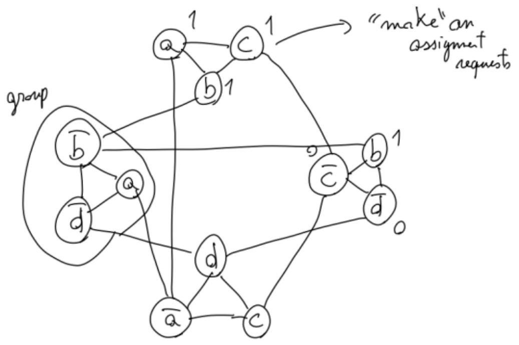
Basically, the presence of an independent set in the constructed graph corresponds to a satisfying truth assignment for the 3SAT instance.

Let's see the main ideas, step by step (figure representing the scenario):

- Pick an arbitrary $3CNF$ Boolean formula f with k clauses (input instance for 3SAT)

$$(a \lor b \lor c) \land (b \lor \overline{c} \lor \overline{d}) \land (\overline{a} \lor c \lor d) \land (a \lor \overline{b} \lor \overline{d})$$

- *Vertices* (the graph is the input instance for Ind Set)
    a. Each vertex represents one literal in f
    b. A *group* of 3 vertices represents a clause (one of the k clauses) – see figure
        i. Assignment request = choose vertices and make a request



- *Edges*:
    a. We add an edge between a literal and its inverse, for all the literals
    b. We add an edge between every pair of vertices that are in the same group

There are two ways to think about 3SAT: (this reasoning coming from here)

- 1. Find a way to assign 0/1 (FALSE/TRUE) to the variables such that the formula evaluates to true, that is each clause evaluates to TRUE
- 2. Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in conflict, i.e., you pick x_i and $\neg x_i$

The reduction works this way:

- The graph will have one vertex for each literal in a clause
- Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

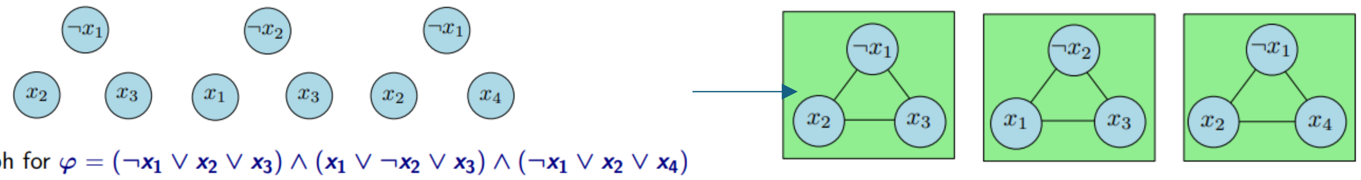
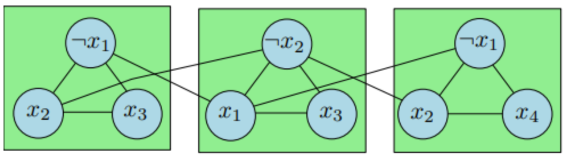


Figure: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

- Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict



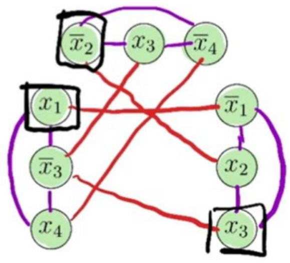
- Take k to be the number of clauses, ensuring they are all "covered"

Remember what *satisfiable* means:

- It asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE

$$\varphi = (x_1 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee x_3 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee x_2 \vee x_3)$$

1. Create a vertex for each literal.
2. Connect each literal to the other two literals in the same clause.
3. Connect each literal x_i to $\overline{x}_i$.



1) *Idea*: independent set represents conflicts ⇒ add an edge between every pair of vertices that are inconsistent (*asking* for opposite assignments to the same variable)
    a. In words: if you choose one vertex, it means it's part of a clause
    b. You have to choose other two vertices which are sure to be different because they are a different independent set

Written by Gabriel R.

c. If you choose one vertex, you <u>have</u> to choose the complement
    i. in order to realize the AND inside the formula

Observation: an independent set with ≥ 1 vertex in each group gives a satisfying truth assignment → should look for indipendent sets of size $\geq k$ to say "YES, f it's satisfiable".

Issue: an independent set now is free to recruit <u>multiple</u> vertices from a group, so I might output "YES, f is satisfiable" even if this not true! ⇒ idea: force the recruitment of <u>one</u> vertex per group (what is represent already in final figure above and the process of said selection drawn on its right).

2) Add one edge between every pair of vertices that one in the same group

This marks the *end of the intuition*. Let's go now into the details of a formal proof.

Claim: G contains an independent set of size exactly $k \Leftrightarrow$ the formula f is satisfiable.

Proof:

1) Suppose f is satisfiable. Pick any satisfying assignment. Each clause in f has ≥ 1 $TRUE$ literal. Thus, we can choose a subset S of k vertices in G that contains exactly one vertex per group such that the corresponding k literals are all $TRUE$. The set S is an independent set because it does not contain both endpoints of any edge of a group, nor of any edge that connects inconsistent literals (as it is derived from a consistent truth assignment)

In simpler terms: a Boolean formula can be made $TRUE$ because we suppose it to be satisfiable (and this works, since vertices are all different). This means we can choose a set of vertices in which we can make every assignment consistent (all different literals and make true each other clause)

2) Suppose G contains an independent set of size k. Each vertex in S must be in a different group. Assign $TRUE$ to each literal of S. Since inconsistent literals are connected by an edge, this assignment is consistent. Since S contains 1 vertex per group, each clause in f contains (at least) one $TRUE$ literal ⇒ f is satisfiable

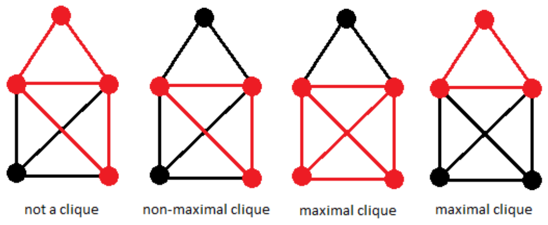
In simpler terms: Having an independent set, you must do a consistent assignment and so you start connecting literals starting from at least one at $TRUE$. Since we have at least vertex per group, this works, given thanks to the independent set, they are all different.

7.6.1 Exercises

These are defined as easy by Scquizzato.

- <u>(Maximum) Clique</u>: compute the longest complete subgraph in G
    a. Other name for a complete graph (from now on, the problem will be called Clique)
    b. Below, a useful figure to clearly see the problem

Theorem: Clique is NP-Hard.

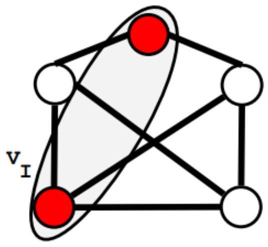


Written by Gabriel R.

<u>Solution</u> (a nice graphical explanation here)

Decision version → $Independent\ Set \leq_p Clique$

- Input: $\langle G = (V, E), k \rangle$
- Output: ∃ in G a clique of size k?



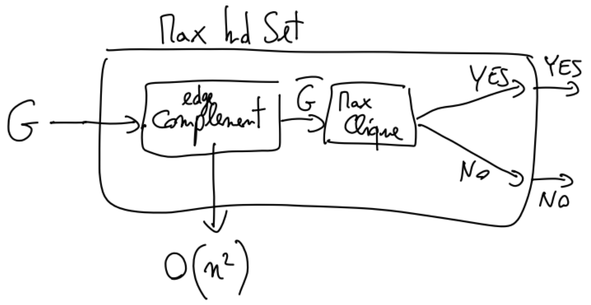
We operate a reduction from Maximum Independent Set (Ham. circuit is not really related to it; as you can see here, one can use 3SAT in order to show Clique is NP-complete). Figure here shows Independent Set.

- *Intuition*
    a. Clique: vertices with <u>all</u> edges between them
    b. Ind. set: vertices with <u>no</u> edges between them

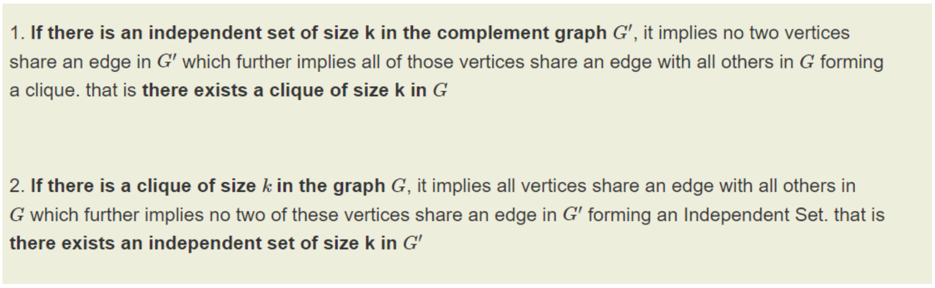
- *Definition*
    a. Given a graph $G = (V, E)$, its <u>edge-complement</u> $\overline{G} = (V, \overline{E})$ has the same vertex V and an edge set $\overline{E}$ such that $(u, v) \in \overline{E} \Leftrightarrow (u, v) \notin E$ (so, no common edges)

- *Observation*
    a. A set of vertices S is independent in $G \Leftrightarrow S$ is a clique in $\overline{G} \Rightarrow$ the largest independent set in G has the same size as the largest clique in $\overline{G}$

To make it super complete, let's draw the schema of what we are doing – takes $O(n^2)$ time, given the constant work needed to traverse all edges *and* vertices:

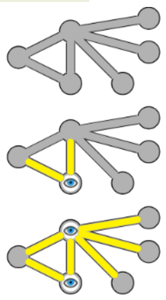


To actually write it formally, so you understand how to prove it, coming from here:


1. **If there is an independent set of size k in the complement graph** G', it implies no two vertices share an edge in G' which further implies all of those vertices share an edge with all others in G forming a clique. that is **there exists a clique of size k in** G

2. **If there is a clique of size** k **in the graph** G, it implies all vertices share an edge with all others in G which further implies no two of these vertices share an edge in G' forming an Independent Set. that is **there exists an independent set of size k in** G'

Definition: a <u>vertex cover</u> of a graph is a set of vertices that includes that at least one endpoint of every edge of the graph



b. Side figure represents such, to be clearer to you

Written by Gabriel R.

Related problem is:

- <u>(Minimum) Vertex Cover</u>: compute the smallest vertex in a given graph
    a. From now on, only called Vertex Cover

Theorem: Vertex Cover is NP-Hard.

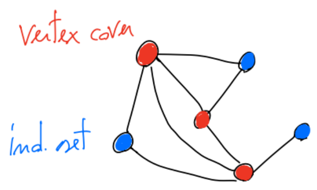
<u>Solution</u> (once again, a nice graphical explanation of this one <u>here</u>)

Decision version → $Vertex\ Cover \leq_p Independent\ Set$

- Input: $\langle G = (V, E), k \rangle$
- Output: $\exists$ in G a vertex cover of size k?

We operate a reduction from Maximum Independent Set (once again, this is the most similar problem to the one we are proving). So, we have $Vertex\ Cover \leq_p Independent\ Set$.

- *Observation*
    a. A set of vertices S is independent in $G \Leftrightarrow V \setminus S$ is a vertex cover of G
        i. In blue there is an independent set (actually the biggest one)
        ii. The other ones are the vertex cover

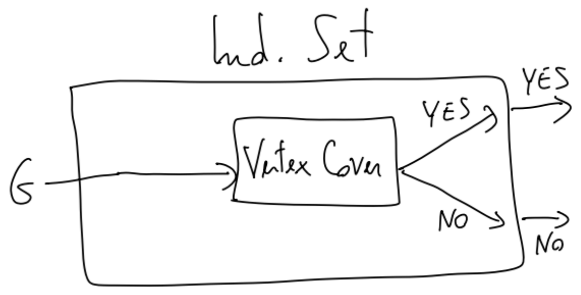


$\Rightarrow$ The longest independent set in G has size $n - k$, where k is the size of the smallest vertex cover of G

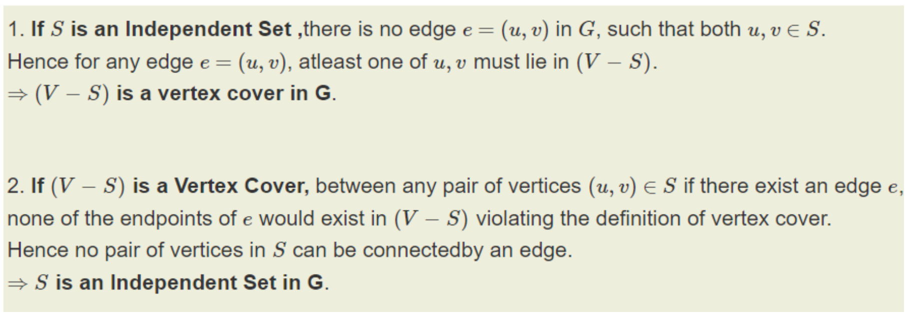
Independent set:

- Input: $\langle G = (V, E), n - k \rangle$
- Output: $\exists$ in G an independent set of size $n - k$?

Once again, let's represent this in a complete way:



To actually write it, follow <u>this</u> one:


1. **If S is an Independent Set** ,there is no edge $e = (u, v)$ in G, such that both $u, v \in S$.
Hence for any edge $e = (u, v)$, atleast one of u, v must lie in $(V - S)$.
$\Rightarrow$ **$(V - S)$ is a vertex cover in G**.

2. **If $(V - S)$ is a Vertex Cover,** between any pair of vertices $(u, v) \in S$ if there exist an edge e,
none of the endpoints of e would exist in $(V - S)$ violating the definition of vertex cover.
Hence no pair of vertices in S can be connectedby an edge.
$\Rightarrow$ **S is an Independent Set in G**.

Exercise

- *Show that*:
    a. $Vertex\ Cover \leq_p Independent\ Set$
    b. $Clique \leq_p Vertex\ Cover$

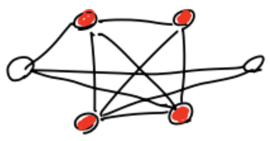
  ⇒ *these 3 problems are equivalent*.

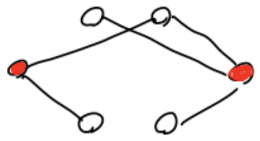
Solution (official = shorter)

- "Same" as $Independent\ Set \leq_p Vertex\ Cover$

- We can consider the following figure for this one
    a. Consider a clique of size 4 in the middle (left)
    b. If you take the complement of this one (right)

- G has a clique of size $k \Leftrightarrow \overline{G}$ has a vertex over of size $n - k$
    a. For the proof: see the book (§ - p. 1106 of 4th edition – theorem 34.12)





Solution (longer and properly explained)

a. Suppose that we have an efficient algorithm for solving Independent Set, it can simply be used to decide whether G has a vertex cover of size at most k, by asking it to determine whether G has an independent set of size at least $n - k$

Given an instance of the Vertex Cover problem, consisting of a graph $G = (V, E)$ and an integer k representing the size, we construct an instance of the Independent Set problem as follows:

1. Let $G' = G$ (i.e., the graph for the Independent Set instance is the same as the original graph G).

2. Let $k' = |V| - k$ (i.e., the target size of the independent set is the number of vertices in G minus the size of the vertex cover k).

To show that this reduction is correct, we need to prove the following:

1. If G has a vertex cover of size $\leq k$, then G' has an independent set of size $\geq k'$.

2. If G' has an independent set of size $\geq k'$, then G has a vertex cover of size $\leq k$.

Let's prove both (1) and (2):

- Suppose C is a vertex cover of size $\leq k$ in G. Then, the set $V \setminus C$ is an independent set in G' (since C covers all the edges, no two vertices in $V \setminus C$ can be adjacent). Furthermore, $|V \setminus C| \geq |V| - k = k'$.

- Suppose S is an independent set of size $\geq k'$ in G'. Then, the set $V \setminus S$ is a vertex cover in G (since S is independent, every edge must have at least one endpoint in $V \setminus S$). Furthermore, $|V \setminus S| \leq |V| - k' = k$.

b. To show that $Clique \leq_p Vertex\ Cover$, we need to provide a polynomial-time reduction from the Clique problem to the Vertex Cover problem. Here's one way to construct the reduction:

Given an instance of the Clique problem, consisting of a graph $G = (V, E)$ and an integer k, we construct an instance of the Vertex Cover problem as follows:

1. Let $G' = G$ (i.e., the graph for the Vertex Cover instance is the same as the original graph G).
2. Let $k' = |V| - k$ (i.e., the target size of the vertex cover is the number of vertices in G minus the size of the clique k).

To show that this reduction is correct, we need to prove the following:

1. If G has a clique of size $\geq k$, then G' has a vertex cover of size $\leq k'$.
2. If G' has a vertex cover of size $\leq k'$, then G has a clique of size $\geq k$.

Proof of (1): Suppose C is a clique of size $\geq k$ in G. Then, the set $V \setminus C$ is a vertex cover in G' (since C is a clique, every edge must have at least one endpoint in $V \setminus C$). Furthermore, $|V \setminus C| \leq |V| - k = k'$.

Proof of (2): Suppose S is a vertex cover of size $\leq k'$ in G'. Then, the set $V \setminus S$ is a clique in G (since S is a vertex cover, every edge must have both endpoints in $V \setminus S$, which means $V \setminus S$ is a clique). Furthermore, $|V \setminus S| \geq |V| - k' = k$.

8 APPROXIMATION ALGORITHMS

These kinds of algorithms are are efficient algorithms that find approximate solutions to optimization problems (in particular NP-hard problems; by hypothesis, $P \neq NP$, otherwise this theory it's useless) with *provable* guarantees on the distance of the returned solution to the optimal one. They solve problems not solvable in polynomial time using approximation.

An *optimization problem* can be described as follows:

$$\Pi: I \times S$$

where $\Pi =$ approximation problem, I = set of inputs and S = set of solutions.

$$c: S \to \mathbb{R}^+$$

Above, the *cost function* c maps each solution to a positive real number.

$$\forall i \in I, S(i) = \{s \in S: i\, \Pi_s\}$$

Above, the the *set of feasible solutions*, and our goal follows.

$$s^* \in S(i)\ and\ c(s^*) = \min \max c(S(i))$$

Here, we want to find the best solution s^* for a minimization/maximization problem. Specifically, we want to find it for the specific instance of that problem $(i \Pi s)$.

8.1 APPROXIMATION

Given a feasible solution $s \in S(i)$, which is OK if $s \neq s^*$, we would want the following:

1) Guarantee on the *quality* of s → approximation factor
    a. By *default* for a *maximization* problem
    b. By *excess* for a *minimization* problem
2) Guarantee on the *complexity*: polynomial-time algorithm

Definition: Let Π be an optimization problem and let A_Π be an algorithm for Π that returns, $\forall i \in I, A_\Pi(i) \in S_i$ (in other words, the choice the algorithm makes). We say that A_Π has an approximation factor (or ratio) of $\rho(n)$ if $\forall i \in I$ such that $|i| = n$ we have (for each one, the concrete translation in problems):

- Minimization problem (basically, an explicit *lower-bound* of the optimal solution)

$$min: \frac{c(A_\Pi(i))}{c(s^*(i))} \leq \rho(n)$$

$$\frac{Greedy}{OPT} \leq \rho(n)$$

So, in a logic of a X-approximation algorithm $\frac{X'}{x^*} \leq \rho$:

a) Upper bound to the cost of X' (which is our solution, greedy choice made by us)

b) Lower bound to the cost of X^* (which is the optimal solution, selected by the algorithm)

As a matter of fact, we have for a min problem:

$$c\big(s^*(i)\big) \geq \frac{c\big(A_\Pi(i)\big)}{\rho(n)}$$

- Maximization problem (basically, an explicit *upper-bound* of the optimal solution)

$$max\colon \frac{c\big(s^*(i)\big)}{c\big(A_\Pi(i)\big)} \leq \rho(n)$$

$$\frac{OPT}{Greedy} \leq \rho(n)$$

As a matter of fact, we have for a min problem:

$$c\big(s^*(i)\big) \leq \rho(n) * c\big(A_\Pi(i)\big)$$

So, in a logic of a X-approximation algorithm $\frac{X^*}{X'} \leq \rho$:

a) Upper bound to the cost of X^* (which is the optimal solution, selected by the algorithm)

b) Lower bound to the cost of X' (which is our solution, greedy choice made by us)

Here, we assume that c maps each feasible solution to a real number $\geq 1 \Rightarrow \rho(n) \geq 1$ always.

The inequality can be easily rewritten as a single expression: $\max\left\{\frac{c(A_\Pi(i))}{c(s^*(i))}, \frac{c(s^*(i))}{c(A_\Pi(i))}\right\} \leq \rho(n)$

- If I have an algorithm which guarantees me I won't pay more than a factor than the optimal solution → this would be very good

Goal: $\rho(n) = 1 + \epsilon$, for the most precise A_Π^* and ϵ as small as possible. *Why do we care? So to choose the problem*. We'll get:

- $\epsilon = 1$ for the Vertex Cover problem
- $\epsilon = \log_2 n$ for the Set Cover problem

Even if we don't know s^*, we are able to estimate the ratio precisely. If the algorithm has an approximation factor of 2, it is called 2-approximation algorithm, for instance.

Much stronger approximation: $\rho(n) = 1 + \epsilon, \forall \epsilon > 0$.

- There exists problems for which we can prove that $\rho(n) = \Omega(n^\epsilon)$
    a. not smaller than n^ϵ $\forall \epsilon < 1$ (e.g., clique)

Definition: An approximation scheme for Π is an algorithm with 2 inputs A_Π (i, ϵ) that $\forall \epsilon$ is a $(1 + \epsilon)$-approximation.

- In this case we just have to choose *how much approximation* we want by tuning the value of ϵ
- In other words: fixed an instance i of size n, the quality is ϵ (whatever ϵ is)

Definition: An approximation scheme is polynomial (PTAS) is $A_\Pi(i, \epsilon)$ is polynomial in $|i|, \forall \epsilon$ fixed. The smaller the value of ϵ, the longer will take the computation (but still polynomial).

(To be pinpoint perfect and formal: we want polynomial schemes both in the input size but also in $\frac{1}{\epsilon}$, which describes it as *fully polynomial*, making the approach generally better).

More notes I collected on the topic.

For approximation algorithms you usually look at the two bounds separately:

- For the upper bound, that is, how far at most the approximated algorithm can get from the optimal one, you will have to do some proving, usually you have to look for some property of the algorithm that maintains some bound during its execution and that then allows you to draw the conclusion
    a. Unfortunately, however, this depends very much on the algorithm itself and there is no general way to find it

This means (for minimization problems):

$$|V'| \leq property$$

- For the lower bound, on the other hand, it is generally always a matter of finding some input of size n (where n must be variable) which always leads to the worst case of the greedy algorithm

This means (for minimization problems):

$$|V^*| \geq property$$

Some <u>*general structure*</u>, studying many problems, I found:

$$GREEDY \leq \; PROPERTY \leq \; OPT$$

Or (using VC = min problem = 2-approx algo)

$$V' \leq property \leq 2|V^*|$$

which is also basically (the greedy is half the property and the optimal is double the property):

$$\frac{V'}{2} \leq property \leq 2|V^*|$$

Usually, anyway, it's enough to know:

$$|V'| \leq 2|V^*|$$

In general, the structure followed by the professor is the following one:

1) Lower bound to the cost of V^*

V^* = This is the absolute best (minimum) solution that can be achieved for the problem. It represents the lowest possible value of the objective function (e.g., cost, length, weight, etc.) that satisfies all constraints of the problem.

The algorithm respects a property. The optimal choice would definitely at least respect that property and then select the least possible. So, in terms of the algo, it's the worst possible choice over the minimum one (so, to select all, and so to respect the general problem property).

Any property or solution provided by the algorithm must respect and cannot surpass this benchmark (V*) in terms of minimization.

2) Upper bound to the cost of V'

V' = This is the solution provided by the greedy or approximation algorithm. While this solution is feasible (i.e., it satisfies the problem's constraints), it may not be the optimal (minimum possible) solution. Instead, it is typically easier to compute and close to the optimal.

This means that, found the general property to respect, the greedy choice made by the algo is at most that one.

Instead, if the problem asks you to

- Prove a general lower bound:

→ Demonstrate that there exist instances where the algorithm performs at the approximation limit, indicating the tightness of the bound

→ Construct an input instance designed to force the algorithm into a worst-case scenario.

➔ $|V'| = 2 * |V^*|$

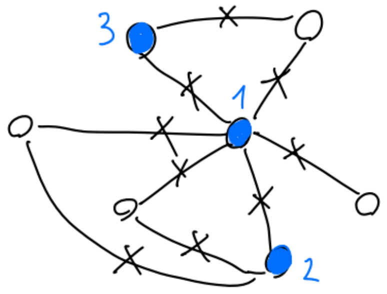
Show one "bad" input instance according to the nature of the problem – that's it.

8.2 Approximation Algorithm for Vertex Cover

Remember Vertex Cover = set of vertices including at least one endpoint of every edge – compute the smallest. What is the very first algorithm you can think about? One way to design such algorithm is using a *greedy approach*:

- Select the vertex with the highest degree
- "Remove" the *touched* edges
- Repeat

Consider the following figure – take 3 as the highest, then 2 and 1 and remove touched edges as said:



Unfortunately, for this algorithm it can be proven that $\rho(n) = \Omega(\log(n))$.

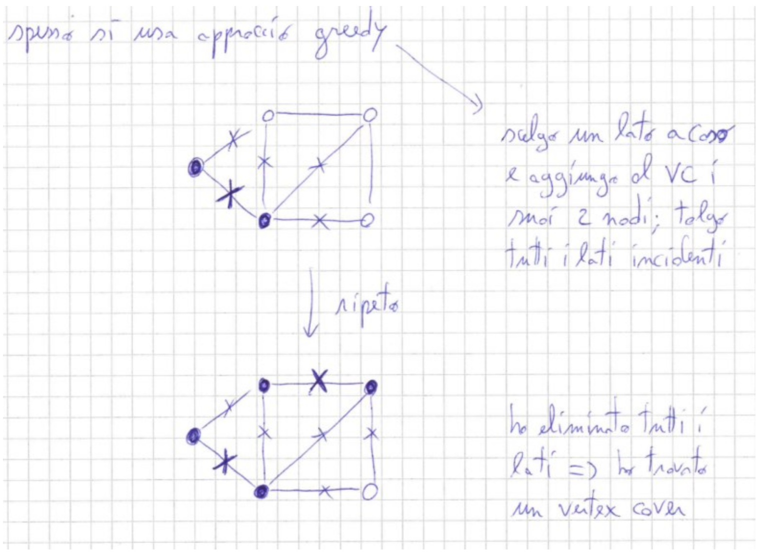
How to prove a LB (Lower Bound)? It's enough to show one "bad" input instance.

<u>Exercise</u>: *show a LB on $\rho(n)$ for this algorithm – the higher, the better* ($\log(n)$ *is difficult*)

(Hint: try to prove the best you can – it should be a constant factor)

We would have to try a new algorithm (again, greedy approach):

- Choose *any* edge
- Add its endpoints to the solution
- "Remove" the covered edges
- Repeat


spesso si usa approccio greedy
scelgo un lato a caso e aggiungo al VC i suoi 2 nodi; tolgo tutti i lati incidenti
ripeto
ho eliminato tutti i lati => ho trovato un vertex cover

We'll show that this is a 2-approximation algorithm (which returns a solution whose cost is at most twice the optimal):

```
procedure $Approx\_Vertex\_Cover(G)$
    $V' = \emptyset$
    $E' = E$
    while $E' \neq \emptyset$: do
        $Let\ (u, v)\ be\ an\ arbitrary\ edge\ of\ E'$
        $V' = V' \cup \{u, v\}$
        $E' = E' \setminus \{(u, z), (v, w)\}$
        $//\ remove\ edges\ that\ have\ u\ and\ v\ as\ endpoints$
    return $V'$
```

Complexity: $O(n + m)$

8.2.1 Analysis

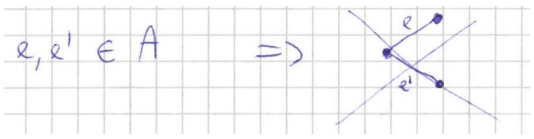
We'll show (because this is a *minimization* problem – remember the structure given above) how to get the quality of the returned solution by the algorithm:

$$\frac{|V'|}{|V^*|} \leq 2$$

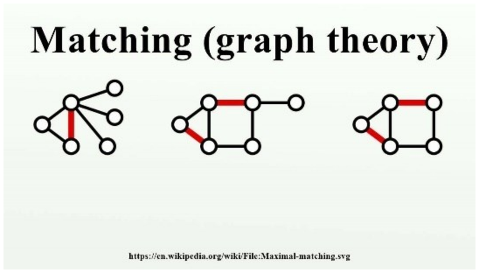
or in inequality terms:

$$|V'| \leq choice \leq 2|V^*|$$

Given A = set of selected edges:

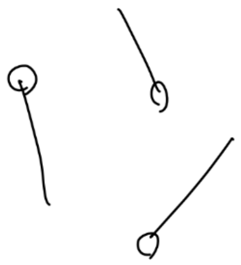


- A is a <u>matching</u>: $\forall e, e' \in A \Rightarrow e \cap e' = \emptyset$
    a. This is also called "independent edge set"
    b. I.e. set of edges with no vertices in common
    c. Every edge is disjoint, so there is no couple of edges sharing a common node
- $Approx\_Vertex\_Cover$ selects a <u>maximal</u> matching: $\forall$ edge $y, A \cup y$ is <u>not</u> a matching
    a. This is a matching which cannot be increased (also maximal $\neq$ maximum)
        i. Not possible to select an edge which touches other vertices



A matching set solves the Vertex cover problem (not in the most efficient way possible, but in a sound way.

On the side, an example of such thing.



Proof:

1. Lower bound to the optimal solution V^*

What can one say about $|V^*|\ vs\ |A|$?

A is a matching ⇒ in V^* there must be ≥ 1 vertex ∀ edge of A (right figure)

In whatever vertex cover, particularly V^*, we have to cover all graph edges and, in particular, all A edges. But A is a matching (so, every edge of A is disjoint), so in V^* there must be at least a vertex for each edge $(u, v) \in A$, given we need to check them all:

$$|V^*| \geq |A|$$

In order to cover the edges in A, any vertex cover - in particular, an optimal cover V^* - must include at least one endpoint of each edge in A. No two edges in A share an endpoint, since once an edge is picked, all other edges that are incident on its endpoints are deleted from E'. Thus, no two edges in A are covered by the same vertex from V^*, meaning that for every vertex in V^*, there is at most one edge in A, giving the lower bound.

2. Upper bound to the algorithm solution V'

What can one say about $|V'|\ vs\ |A|$?

- $|V'| \leq 2|A|$, or even better $|V'| = 2|A|$ by construction and so:

$$(1. + 2.) \Rightarrow |V'| \leq 2|A| \leq 2|V^*| \Rightarrow \rho(n) = \frac{|V'|}{|V^*|} \leq 2$$

This concludes the proof: $Approx\_Vertex\_Cover$ is a 2-approximate algorithm for $Vertex\ Cover$. Also:

- It can never be that this algorithm returns a set of nodes twice as bad as the minimal set
- It can never be larger than twice the theoretical minimum set
- It is not possible to improve this approximation (*tight* approximation)
    a. That depends on your definition of approximation ratio
    b. Normally the approximation ratio is defined as the worst ratio between optimal solution and the one produced by your algorithm
    c. If this is the case, all you need to show that the ratio is tight is come up with one bad example, which shows it works for all sizes

Specifically, if the ratio is 2 we have a 2-approximation algorithm, meaning it gives solutions that never cost more than twice that of optimal if it is a minimization problem, or never provide less than half the optimal value if it is a maximization problem.

This is a common strategy in approximation proofs: we don't know the size of the optimal solution, but we can set a lower bound on the optimal solution and relate the obtained solution to this lower bound.

Exercise: *show that the approximation factor of $Approx\_Vertex\_Cover$ is exactly 2*

State of the art of this problem:

- $\exists\ 2 - \theta(\frac{1}{\sqrt{\log(n)}})$ approximation algorithm
- Vertex cover cannot be approximated better than ~ 1.36
- Conjecture: cannot be approximated better than 2

Keep in mind what the CLRS says.

- At first, you might wonder how you can possibly prove that the size of the vertex cover returned by $Approx\_Vertex\_Cover$ is at most twice the size of an optimal vertex cover, when you don't even know the size of an optimal vertex cover.
- Instead of requiring that you know the exact size of an optimal vertex cover, you find a *lower* bound on the size.

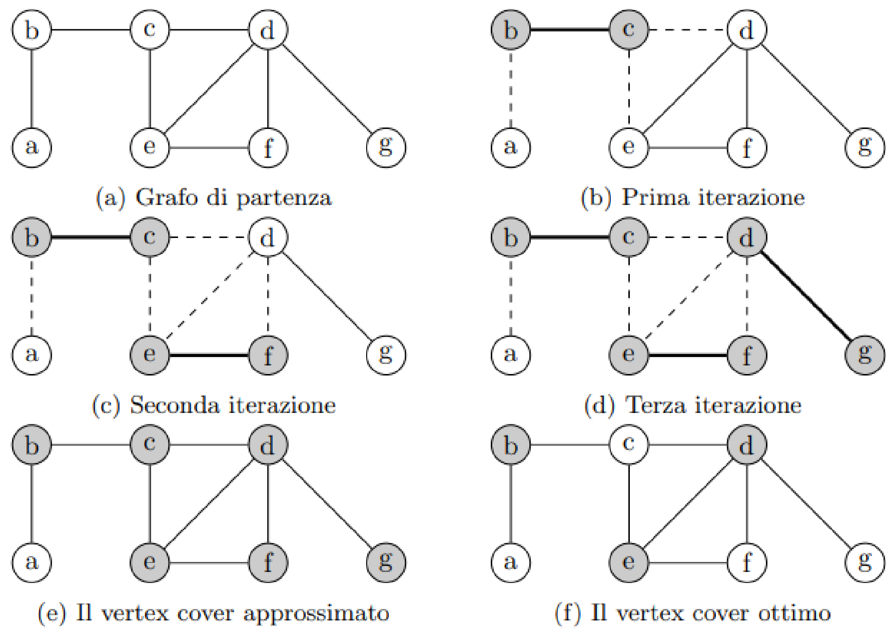


Figura 1.1: Esempio grafico di applicazione dell'APPROX_VC

8.2.2 Exercises

<u>Exercise</u>: *show a LB on* $\rho(n)$ *(so,* 2 *I add) for this algorithm – the higher, the better* ($\log(n)$ *is difficult*)

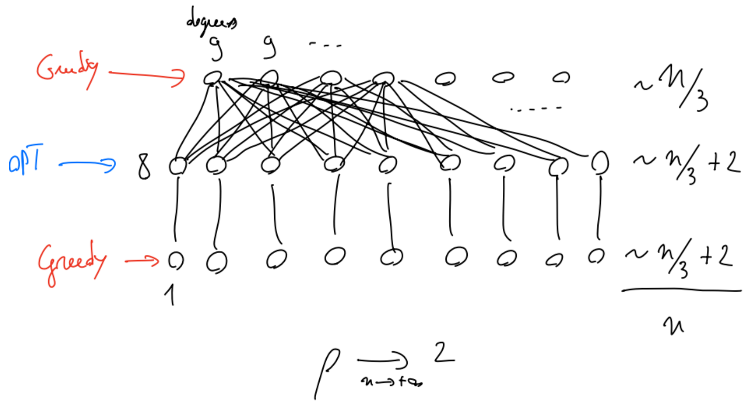
(Hint: try to prove the best you can – it should be a constant factor)

<u>Solution</u>

One possible idea is the following:

- take a round of vertices
- consider levels of vertices adding more

We call this problem: "degree-based greedy approximation for vertex cover". Consider the following:



Written by Gabriel R.

This image demonstrates a general idea for constructing a "bad" input instance to show a lower bound on the approximation ratio. The approach is to create a graph with multiple levels, where each level has more vertices than the previous level, but with fewer edges connecting to the next level.

The reasoning goes like this:

- Start with a single vertex (labeled "Greedy" in the image)
- At the next level, add a few vertices (e.g., 3) that are all connected to the first vertex
- At the next level, add more vertices (e.g., 8)
    - that are only connected to the previous level vertices
- Continue adding more and more vertices at each level
    - with fewer connections to the previous level

The idea is that the greedy algorithm will pick all the vertices in the first level, then all the vertices in the second level, and so on, resulting in a large vertex cover. However, the optimal vertex cover would be to pick the intermediate level vertices, which can cover all the edges with fewer vertices.

The greedy algorithm, by design, will select the vertex with the highest degree at each step. This means:

- It will select the single vertex at the first level.
- Then, it will select all $\frac{n}{3} + 2$ vertices at the second level (since they have the highest degree at that point).
- Next, it will select all $\frac{n}{3} + 2$ vertices at the third level (since they are now the highest degree vertices remaining).

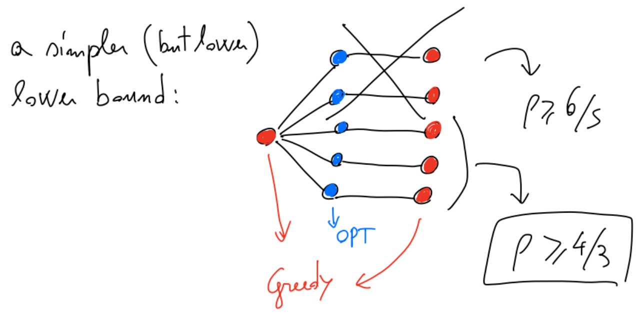
Therefore, the total size of the vertex cover produced by the greedy algorithm is: $1 + \left(\frac{n}{3} + 2\right) + \left(\frac{n}{3} + 2\right) = \frac{2n}{3} + 5$

However, the optimal vertex cover for this graph is to select the $\frac{n}{3} + 2$ vertices at the second level. This covers all edges in the graph using only $\frac{n}{3} + 2$ vertices.

By comparing the greedy solution size ($\frac{2n}{3} + 5$) to the optimal solution size ($\frac{n}{3} + 2$), we get an approximation ratio of:

$\rho = (\frac{2n}{3} + 5) \,/\, (\frac{n}{3} + 2) \approx 2$ (for large values of n)

The following considers a simpler idea instead:



1. In a bipartite graph $G = (U, V, E)$, where U and V are the two disjoint vertex sets, and E contains edges only between U and V
2. Consider the vertex v in U that has the maximum degree (i.e., connected to the most vertices in V)
3. The greedy algorithm will select v and all its neighbors in V
4. However, the optimal solution is to select only the neighbors of v in V (and not v itself)
5. This gives a lower bound on the approximation ratio $\rho \geq (1 + deg(v))\ /\ deg(v) = 1 + 1/deg(v)$

The key observation is that by selecting the highest degree vertex v in U, the greedy algorithm is making the worst possible choice compared to the optimal solution of just selecting v's neighbors in V. This lower bound holds because:

- Greedy picks v and $deg(v)$ vertices in V, so size is $1 + deg(v)$
- Optimal just picks the $deg(v)$ vertices in V that are neighbors of v

So the approximation ratio is at least $(1 + deg(v))\ /\ deg(v)$, which approaches $1 + 1/deg(v)$ as $deg(v)$ grows large.

- So, we show a bad input instance; instead of directly choosing the neighbors, we choose the highest degree vertices
    a. which when running $Approx\_Vertex\_Cover$ would select iteratively edges
        i. then removing the incidents ones
    b. The selection would be suboptimal, given it would only have to select the neighbors

Exercise: *show that the approximation factor of $Approx\_Vertex\_Cover$ is exactly 2.*

Solution

OPT: just one vertex

Consider the algorithm:

- The algorithm starts with an empty set V' (vertex cover set) and the original edge set E'. It iteratively selects an arbitrary edge (u, v) from E' and adds both vertices u and v to the vertex cover set V'. It then removes all edges from E' that are incident on either u or v
- The algorithm continues this process until E' becomes empty, meaning all edges have been covered by the selected vertices in V'. Finally, it returns V' as the approximate vertex cover

The key observation is that for each edge (u, v) selected, at least one of u or v must be present in the optimal vertex cover OPT. This is because OPT must cover all edges, and (u, v) is an edge in the original graph.

Therefore, during each iteration when an edge (u, v) is processed, the algorithm adds at most two vertices to V', while the optimal vertex cover OPT must contain at least one of these two vertices.

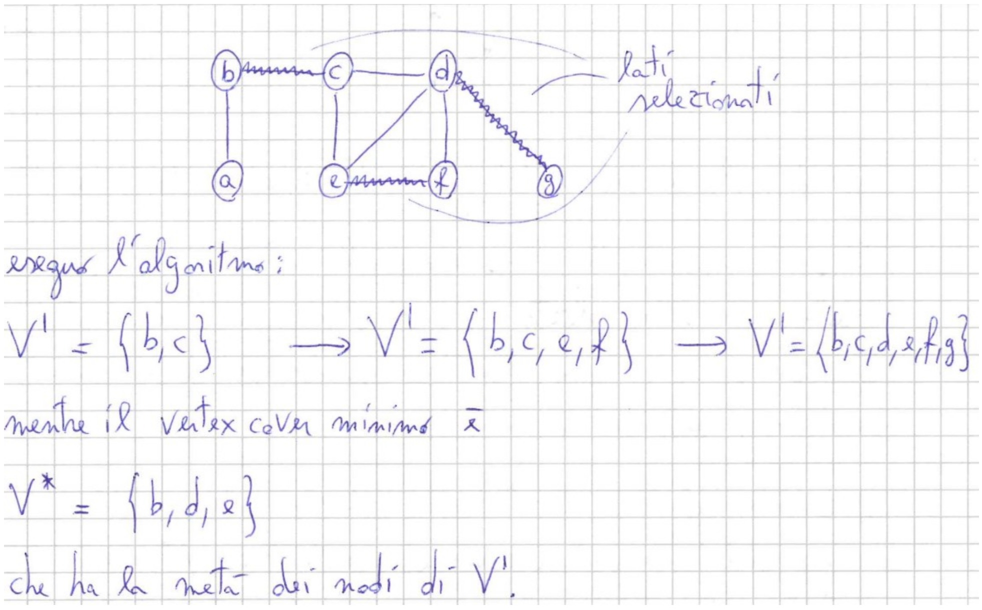
Consequently, we can establish the following inequality:

$$|V'| \leq 2 * |OPT|$$

The bound is tight; ensuring the greedy choice is 2 vertices and the optimal choice is just one vertex, we will have that $\frac{|V'|}{|OPT|} = 2 \leq 2$.

(Also, from past years of notes in the italian version of this course)

No, it can't be improved: this algorithm can, in some cases, return exactly twice the number of minimal nodes.


b
c
d
a
e
f
g
lati selezionati

esegue l'algoritmo:

$V' = \{b, c\} \longrightarrow V' = \{b, c, e, f\} \longrightarrow V' = \{b, c, d, e, f, g\}$

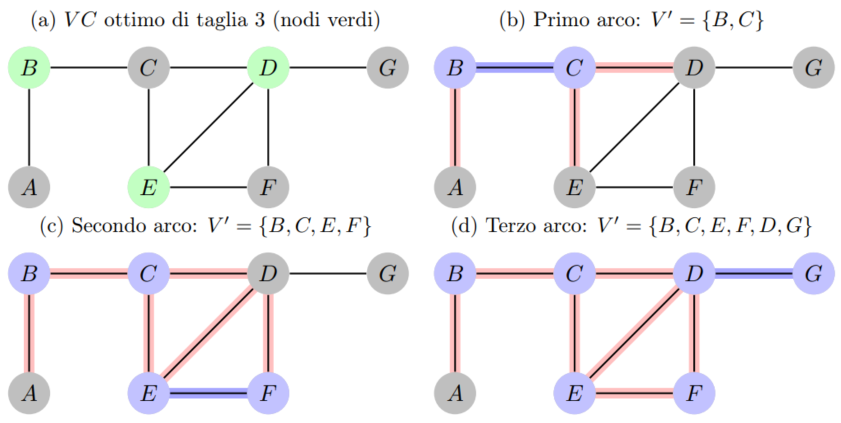
mentre il vertex cover minimo è

$V^* = \{b, d, e\}$

che ha la metà dei nodi di V'.

Another example is the following one, in which the order of choice of edges is unlucky, always a multiple of 2 given the minimum size, in this case 3.

Figura 3.1: Esempio di esecuzione pessima dell'algoritmo di approssimazione. Gli archi blu sono quelli presi in esame all'iterazione corrente, gli archi rossi sono quelli coperti dai nodi selezionati. I nodi blu sono quelli selezionati in V'.



(a) VC ottimo di taglia 3 (nodi verdi)

(b) Primo arco: $V' = \{B, C\}$

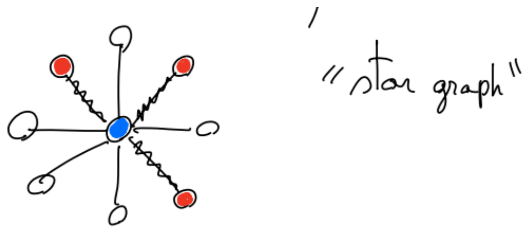
(c) Secondo arco: $V' = \{B, C, E, F\}$

(d) Terzo arco: $V' = \{B, C, E, F, D, G\}$

Exercise: *modify $Approx\_Vertex\_Cover$ so as to select only one vertex instead of both of them. How would ρ become?*

(Note: in the CLRS this is also called “Give an example of a graph for which $Approx\_Vertex\_Cover$ always yields a suboptimal solution”)

Solution



$OPT = 1$

$ALG = n-1$ $\rightarrow \rho \geq n-1$

Consider the *star graph*, a bipartite graph with one internal node (given n vertices) and $n-1$ leaves. The optimal choice would select one vertex then the greedy selects the leaf nodes. This would imply removing all edges connected to the intermediate node and, as such, we guarantee to select one vertex at a time, ensuring $\rho \geq n-1$. Selecting only one vertex can be really bad unless you trick the algorithm a bit.

In this structure:

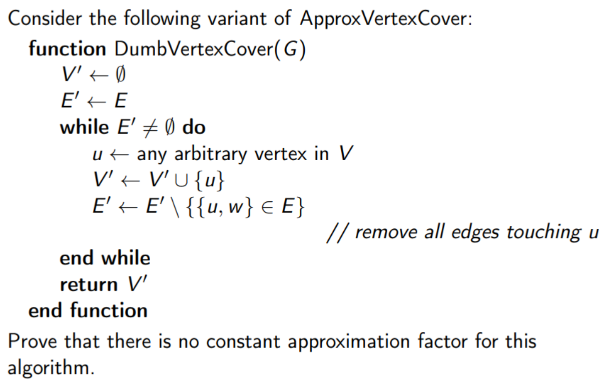
1. The optimal vertex cover (OPT) contains only the central vertex, covering all $n-1$ edges. So $OPT = 1$.
2. The modified approximation algorithm selects one endpoint vertex per edge. For the star graph, this means it will select all $n-1$ leaf vertices.
3. Therefore, the size of the approximate vertex cover produced by the algorithm is $|C| = n-1$.
4. Since $OPT = 1$ and $|C| = n-1$, the approximation ratio $\rho = |C| \,/\, OPT = (n-1) \,/\, 1 = n-1$.

So for the star graph, the approximation ratio ρ achieved by the modified algorithm is exactly $n-1$, which matches the lower bound claim of $\rho \geq n-1$ in the image.

More completely (looking online):

- the $Approx\_Vertex\_Cover$ algorithm works by iteratively selecting an edge, adding both its endpoints to the vertex cover, and removing all the edges incident to these two vertices
- in a star graph, the algorithm would end up selecting all the leaf nodes along with the central node, yielding a suboptimal solution
    - as the optimal solution would only include the central node

Exercise (done in previous years)

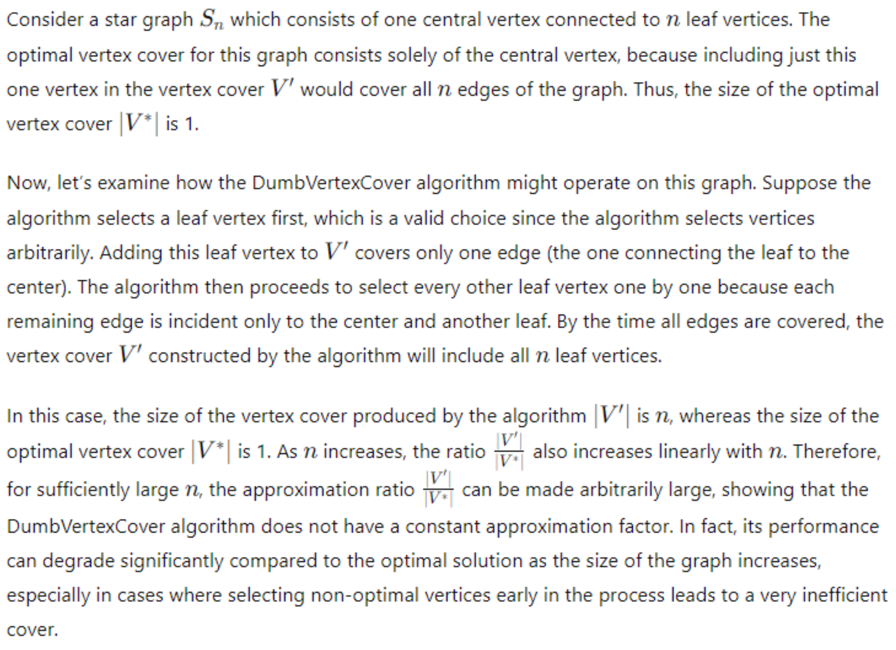


Consider the following variant of ApproxVertexCover:

```
function DumbVertexCover($G$)
    $V' \leftarrow \emptyset$
    $E' \leftarrow E$
    while $E' \neq \emptyset$ do
        $u \leftarrow$ any arbitrary vertex in $V$
        $V' \leftarrow V' \cup \{u\}$
        $E' \leftarrow E' \setminus \{\{u, w\} \in E\}$
                                        // remove all edges touching $u$
    end while
    return $V'$
end function
```

Prove that there is no constant approximation factor for this algorithm.

As always, the counterexample is the star graph:



Consider a star graph S_n which consists of one central vertex connected to n leaf vertices. The optimal vertex cover for this graph consists solely of the central vertex, because including just this one vertex in the vertex cover V' would cover all n edges of the graph. Thus, the size of the optimal vertex cover $|V^*|$ is 1.

Now, let's examine how the DumbVertexCover algorithm might operate on this graph. Suppose the algorithm selects a leaf vertex first, which is a valid choice since the algorithm selects vertices arbitrarily. Adding this leaf vertex to V' covers only one edge (the one connecting the leaf to the center). The algorithm then proceeds to select every other leaf vertex one by one because each remaining edge is incident only to the center and another leaf. By the time all edges are covered, the vertex cover V' constructed by the algorithm will include all n leaf vertices.

In this case, the size of the vertex cover produced by the algorithm $|V'|$ is n, whereas the size of the optimal vertex cover $|V^*|$ is 1. As n increases, the ratio $\frac{|V'|}{|V^*|}$ also increases linearly with n. Therefore, for sufficiently large n, the approximation ratio $\frac{|V'|}{|V^*|}$ can be made arbitrarily large, showing that the DumbVertexCover algorithm does not have a constant approximation factor. In fact, its performance can degrade significantly compared to the optimal solution as the size of the graph increases, especially in cases where selecting non-optimal vertices early in the process leads to a very inefficient cover.

Exercise (done in previous years)

$G \rightarrow G^c$ contains all edges not in G. V^*: vertex cover of $G \Rightarrow V \setminus V^*$ is a clique of maximum size in G^c. Is it possible to approximate CLIQUE using a 2-approx algorithm for Vertex Cover?

Solution

No (reductions between problem do not preserve the approximation factor)

$$G \rightarrow |C_{max}| = \frac{n}{2}$$

in G^c, $|V^*| = n - \frac{n}{2} = \frac{n}{2}$.

Apply $Approx\_VC(G^c) \rightarrow V'$ it could be that $|V'| = n$ that is $2|V^*|$

$\Rightarrow$ returns a clique of size $n - n = 0$

$\Rightarrow$ no approx for no ρ

In other words: this works if the vertex cover is of maximum size, but the transformation does not preserve the approximation.

Written by Gabriel R.

9 TSP & Metric TSP

(Further readings on this one: here)

9.1 Travelling Salesperson Problem (TSP)

Definition: Given a complete, undirected ($c(u,v) = c(v,u)$ = symmetric) graph $G = (V,E)$ and a function $w: E \to \mathbb{R}^{+}$, output a tour $T \subseteq E$ (i.e. a cycle that passes through every vertex exactly once) minimizing $\sum_{e \in T} w(e)$.

The problem answers to: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

Collectively:

$$T \subseteq E \text{ minimizing } \sum_{e \in T} w(e)$$

- $w: E \to \mathbb{R}^{+}_{0}$: we will work only on positive weights
    a. We can do this without loss of generality (wlog) because every TSP tour has the same number of edges ⇒ we can add a large weight to each edge, such that edges have non-negative weights

Theorem: For any function $\rho(n)$ that can be computed in time polynomial in n, there is no polynomial-time $\rho(n) -$ approximation algorithm for TSP with $\rho = O(1)$, unless $P = NP$.

Proof: We will use a reduction from the Hamiltonian Circuit, so $Ham \leq_p TSP$. The reduction function is $f: \langle G = (V,E) \rangle \to \langle G^K = (V, E^K), c \rangle$. This is a generic instance which we want to reduce to: a simple graph and not necessarily complete. So, given $n = |V|$, build the following reduction:

- $G \to G' = (V, E')$ complete
- $w(e \in E') = \begin{cases} 1, & e \in E \\ \rho n + 1, & otherwise \end{cases}$
    a. *Idea*: weights are far apart
    b. 1 means "adding weight 1 to all edges that were there before"

Now, actually proving the theorem:

1) If G has an Hamiltonian circuit ⇒ ∃ a tour of cost n ⇒ TSP algorithm's run on G' returns a tour of cost $\boxed{\leq \rho n}$ (so, there's a cycle passing through all edges and its max cardinality can be the number of vertices)

2) If G has no Hamiltonian circuit ⇒ the TSP algorithm run on G' returns a tour of cost

$(\rho n + 1) + (n - 1) = \rho n + 1 \boxed{> \rho n}$

Thus, if we could approximate TSP within a factor of ρ in poly-time, then we would have a poly-time algorithm for Hamiltonian Circuit.

- So, suppose that we apply the approximation algorithm A on G'
    a. Because A is guaranteed to return a tour of cost no more than ρ times the cost of an optimal tour, if G contains a Hamiltonian cycle, then A must return it
    b. If G has no Hamiltonian cycle, then A returns a tour of cost more than ρn. Therefore, we can use A to solve the Hamiltonian-cycle problem in polynomial time

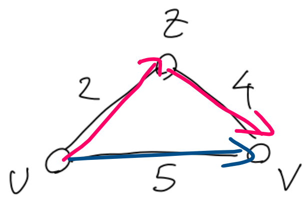
So, we've shown NP-Hardness not for the original problem, but for the approximate version of the problem.

9.2 Metric TSP

Metric TSP is a special case of TSP where the weight function w satisfies the *triangle inequality*:

$$\forall\, u, v, z \in V, \text{ it holds that } w(u,v) \leq w(u,z) + w(z,v)$$

The following is an example of that:



This inequality is satisfied for geometric graphs, where the vertices are points in the plane (or some higher-dimensional space), edges are straight line segments, and lengths are measured in the usual Euclidean metric.

So, the path from u to v is shorter than any other path that passes through another vertex (more convenient than using (u,z) and (z,v) edges (travel *directly* rather than *indirectly*).

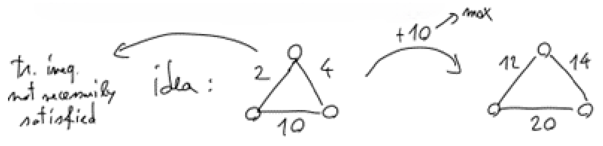
$\Rightarrow c(\langle u,v\rangle) \leq c(\langle u,w,v\rangle)$ (this problem was also called TRIANGLE_TSP in old Italian years)

- Is $Metric\ TSP$ in P? (often special cases are in P – for the curious ones of you, here)

Theorem: $Metric\ TSP$ is NP-Hard

Proof: $TSP \leq_p Metric\ TSP$ - the idea is to multiply the weight of every edge by the value of the heaviest edge of the graph.

The idea is the following (where inequality is not strictly satisfied):



Given an instance of the TSP problem $\langle G = (V,E), w, k\rangle$, we build an instance of Metric TSP $\langle G' = (V,E), w', k'\rangle$ such that the triangle inequality is satisfied in G'. In order to to this, we can define the *weight* function $w\,'$ as follows:

$$w'(u,v) = w(u,v) + W$$

$$\text{giving } W = \max_{u,v \in V}\{w(u,v)\}$$

$$\langle G=(V,E), w, k\rangle$$
$$\rightsquigarrow$$
$$\langle G'=(V,E), w', k'\rangle$$
$$w'(u,v) = w(u,v) + \underset{\max\limits_{u,v\in V}\{w(u,v)\}}{\overset{\parallel}{W}}$$

Written by Gabriel R.

Think of a value k' in such a way there if there exist an Hamiltonian circuit, there will be one in G' in such a way the cost of the tour will work for every edge, so:

$$k' = k + nW$$

9.2.1 NP-Hardness of Metric TSP

To be shown yet:

1) w' satisfies triangle inequality
2) $\exists$ an Hamiltonian circuit of cost k in $G \Leftrightarrow \exists$ Hamiltonian circuit of cost k' in G'

Let's see how to solve them formally:

1) $w'(u,v) \leq^? w'(u,w) + w'(w,v)$ (is it at most the weight of the others)?
   $w(u,v) + W \leq^? w(u,w) + w(w,v) + 2W$ (does this hold adding a general weight)?
   $w(u,v) \leq^? w(u,w) + w(w,v) + W$ (simply adding W both members)
   $\underbrace{w(u,w)}_{\geq 0} + \underbrace{w(w,v)}_{\geq 0} + \underbrace{W - w(u.v)}_{\geq 0} \geq^? 0$ (is it true this is at most 0)?

   We only ask if the definition of triangle inequality is satisfied correctly.
   Note: it's important the weights of edges are non-negative (otherwise, last part does not hold).

2)
   a. ($\Rightarrow$) $\exists$ Ham. circuit of cost k in G. Note that an optimal solution contains exactly n edges and the same circuit in G' introduces a weight for every edge (so, $+W$ $\forall$ edge). Thus, the cost of said tour in G' is $k + nW$.
   b. ($\Leftarrow$) just remove the $+W$ $\forall$ edge to obtain a Hamiltonian circuit of cost k in G.

So, what this means is (from CLRS): the traveling-salesperson problem is NP-complete even if you require the cost function to satisfy the triangle inequality. Thus, you should not expect to find a polynomial-time algorithm for solving this problem exactly.

9.3 2-Approximation Algorithm for Metric TSP

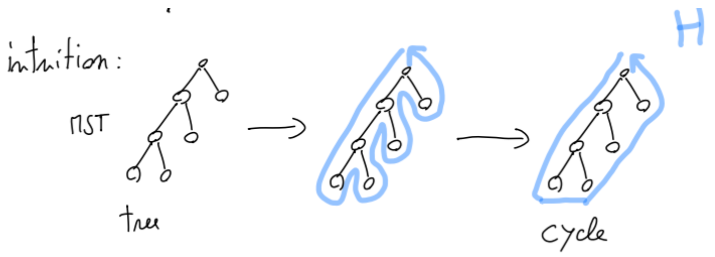
For the Vertex Cover problem we used the concept of maximal matching to find a 2-approximation algorithm (basically $Vertex\ Cover \rightarrow Matching$). That means, we can find a tour that is no longer than twice the shortest tour. See this one here.

What is the most similar problem to $Metric\ TSP$? MST (Minimum Spanning Tree). We simply travel "around" the MST , using each edge twice and the trip is shorter since the triangle inequality holds. We give the following *intuition*:

- We give an MST
- We want to build a *cycle*: what to do on a tree to achieve it?
- Basically, there is a DFS traversing all the nodes

Vertex cover $\rightsquigarrow$ matching
metric TSP $\rightsquigarrow$ MST

- The cycle forms having all nodes touched exactly once



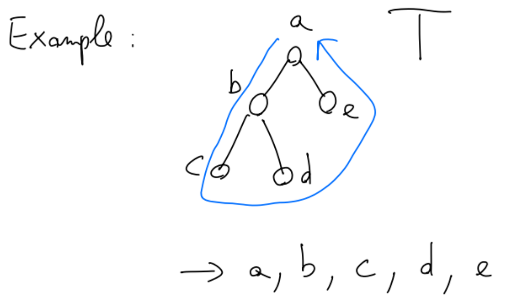
The total MST weight gives a lower bound on the length of an optimal TSP tour. We shall then use the MST to create a tour whose cost is no more than twice of the MST's weight, as long as the cost function satisfies the triangle inequality. Obviously, the optimal value for a tour is at least the value of the MST, since the tour itself spans the graph and is acyclic.

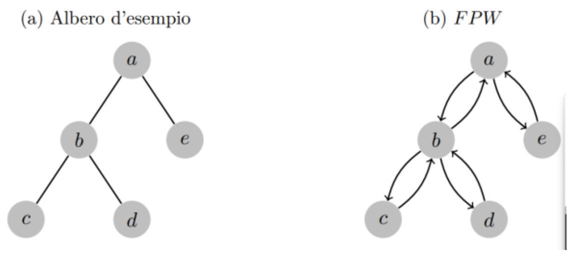
We use MST computing said triangle inequality, but asking ourselves the question "tree – cycle"?

- How to transform the tree in a cycle?
- In order to do this, we can use the *preorder* visit of the MST

```
procedure $PREORDER(T, v)$
    $print(v)$
    if $interval(v)$: do
        for each v ∈ children(v): do
            $PREORDER(v)$ //simple recursive call
    $return$
```

Let's depict an example here:



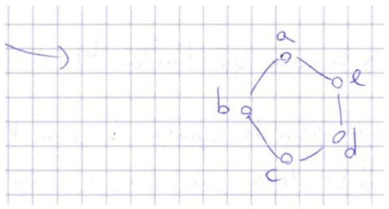


The process of preorder traversal can be represented as "root – left – right".

Idea: add to the preorder list the root (to close the cycle) → Hamiltonian cycle of the original graph

Note the result is <u>not</u> a cycle since e and a are not connected. We can simply solve this problem by adding the edge (e, a) to the $PREORDER$ list and make it an Hamiltonian circuit.

We are free to add every edge we want because the graph is <u>complete</u> by definition. The resulting cycle can be seen from the direction of the blue arrow above.



To do so, we define the following algorithm:

```
procedure $Approx\_Metric\_TSP(G)$:
    $V = \{v_1, v_2, \dots v_n\}$
    $r = v_1$ $//root\ from\ which\ Prim\ is\ run$
    $T^* = Prim(G, r)$ $//\ compute\ an\ MST\ T\ from\ G\ from\ the\ root\ r$
    $\langle v_{i_1}, v_{i_2}, \dots v_{i_n} \rangle\ = H' = PREORDER(T^*, r)$
    $//\ lists\ all\ the\ vertices\ in\ the\ tree\ in\ an\ ordered\ fashion\ following\ a\ preorder\ walk$
    return $\langle H', v_{i_1} \rangle \geq H$ $//\ basically, close\ the\ cycle\ and\ return\ it$
```

This algorithm uses Prim as a subroutine to compute the MST. As such, this is super-fast and can be characterized as a near-linear algorithm.

So, to fully summarize:

1. Given a complete weighted graph G, pick any vertex v as the root and find a MST T, using Prim's algorithm
2. Compile a list L of vertices encountered in a preorder traversal of T
3. Return L as a tour

Since L contains each vertex exactly once, it constitutes a tour where there's an edge between each pair of consecutive vertices, and between the first and the last vertex.

To clearly see the algorithm, consider this one here.

Consider the following from the book – also present in other notes existing, so given it's useful I put this here too:

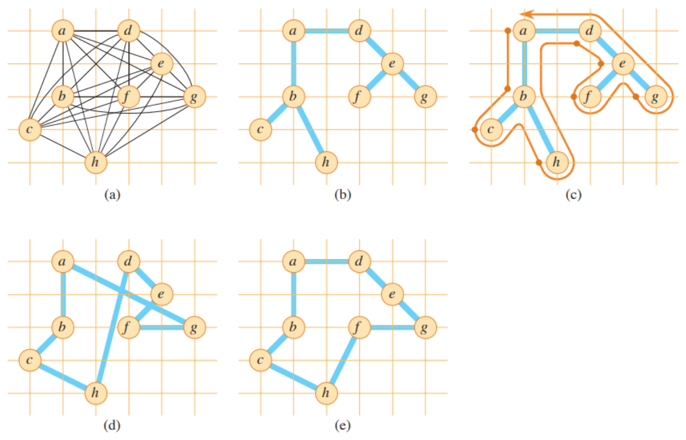


Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g. **(d)** A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour H^* for the original complete graph. Its total cost is approximately 14.715.

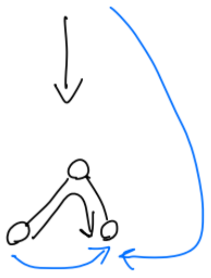
9.3.1 Analysis of the cost of H

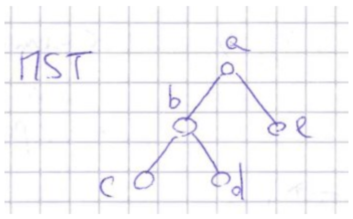
Let H^* denote an optimal tour for the given set of vertices.

Let's give the intuition behind the algorithm:

1) Cost of T^* is "low" (actually, the lowest)
2) Triangle inequality ⇒ "shortcuts" do not increase the cost

For example, in the graph below, to go from c to d in the MST tree, we don't go from b.





Shortcutting allows to construct a tour which does not revisit vertices. Specifically, it can be formally defined as follows (considering π is the parent if you remember):

$$\pi = \langle \cdots, u, z, v, \cdots \rangle \rightsquigarrow \pi' = \langle \cdots, u, v, \cdots \rangle$$

$$c(\pi') \leq c(\pi)$$

To be more precise, shortcuts in graph theory refer to edges that directly connect two vertices that are not adjacent in the original graph. What we are trying to do here is:

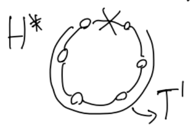
$$\frac{w|H^*|}{w|H'|} \leq 2 \quad or\ more\ in\ general\ \ \rho = \frac{|H'|}{|H^*|}\ (not\ considering\ weights\ properties)$$

or in inequality terms:

$$w|H'| \leq choice \leq 2w|H^*|$$

1) Lower bound to the cost of H^* (= optimal tour) (for vertex cover: $|V^*| \geq |A|$)

$$w(H^*) \geq ?$$



$$w(H^*) \overset{\nearrow \text{ weights are } \geq 0}{\geq} w(T')$$

$$T' \text{ is a spanning tree}$$

$$\Downarrow$$

$$w(T') \geq w(T^*)$$

Given the optimal tour, we obtain a spanning tree by deleting any edge from a tour, and each edge cost is non-negative (so, its cost is not lower than the best spanning tree found). Also, the triangle inequality tells us that the cost of taking this edge is at least as short as the original in-order path.

Therefore, the weight of the MST T^* provides a lower bound to the cost of an optimal tour, so we have $w(T) \leq w(H^*)$.

2) Upper bound to the cost of H (the returned solution). We want to prove the following:

$$\underline{w(H) \le \boxed{\rho w(T^*)}} \le \rho w(H^*)$$

$$\uparrow \text{ comes from}$$

$$w(H^*) \ge w(T^*)$$

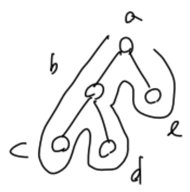
The approximation factor keeps being at most twice, so $\rho = 2$ (where $w(T^*)$ has to be at least twice as big since by last line would not hold otherwise):

$$w(H) \le 2w(T^*) \text{ ?}$$

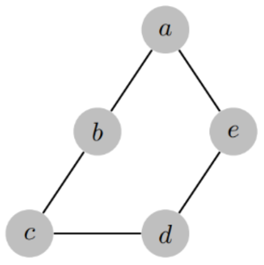
Definition: given a tree, a full preorder chain is a list with repetitions of the vertices of the tree which identifies the vertices reached from the recursive calls of $PREORDER(T, v)$.

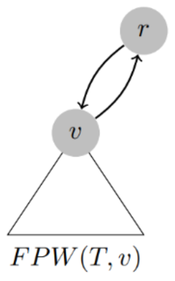
The following is an example, quite easy to see I hope (f.p.c. = "full preorder chain" from now on):

Example :



f.p.c. : a, b, c, b, d, b, a, e, a





As a sidenote, consider the *full walk*, which would list the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. The full preorder walk allows to select each edge of the MST exactly twice, one going down, the other going up, as you can see from right figure.

The key property is the following: given the full preorder chain traverses every edge exactly two times, we have:

$$w(f.p.c.) = 2w(T^*)$$

This happens because every edge of T^* appears twice in a f.p.c.

Unfortunately, the f.p.c. is *generally not a tour* since it visits some vertices *more than once*.

- By the triangle inequality, however, we can delete a visit to any vertex from f.p.c. and the cost does not increase
    a. This ensures we only have traversed vertices twice so to ensure a full visit in all the tree, correctly applying the Hamiltonian cycle definition
- By repeatedly applying this operation, we can remove from f.p.c. all but the first visit to each vertex (except for the second [and last] occurrence of the root)
- This is like adding a *shortcut* between vertices that *does not increase the cost*

Consider the cycle being returned is a subsequence of the full preorder walk, selected the first time they are seen. Thanks to the shortcutting property, we remove the duplicates:

$$\begin{aligned}
& 2w(T^*) = w(\langle a, b, c, \cancel{b}, d, b, a, e, a \rangle) \\
& \geq w(\langle a, b, c, d, \cancel{b}, a, e, a \rangle) \quad \text{shortcut} \\
\text{triangle ineq.} \quad & \geq w(\langle a, b, c, d, \cancel{a}, e, a \rangle) \\
& \geq w(\langle a, b, c, d, e, a \rangle)
\end{aligned}$$

$\Rightarrow 2w(T^*) \geq w(H)$

Putting all pieces together:

1) $w(H^*) \geq w(T^*)$

2) $2w(T^*) \geq w(H)$

$$2w(H^*) \geq \boxed{2w(T^*)} \geq w(H)$$

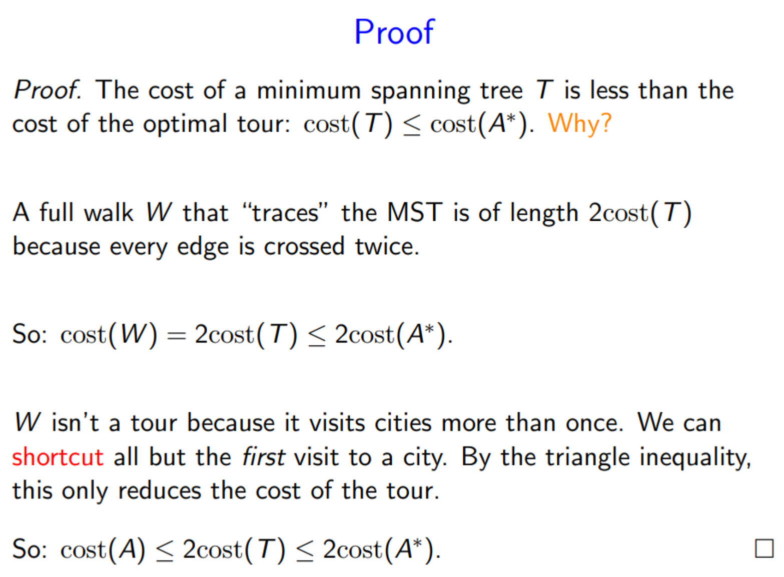
$$\Rightarrow \frac{w(H)}{w(H^*)} \leq 2$$

(This structure holds, compared to the general one, since the inequalities are inverted in sign and logic. This is given by the particular structure of the problem, otherwise the structure above works)

This ordering is the same as that obtained by a preorder walk of the tree T . Let H be the cycle corresponding to this preorder walk. It is a Hamiltonian cycle, since every vertex is visited exactly once, and in fact it is the cycle computed by $Approx\_Metric\_TSP$. This algorithm is not the best practical choice for this problem. There are other approximation algorithms that typically perform much better in practice.

So, basically:

- A valid tour must span all the vertices
- When we perform a preorder traversal of the MST, each edge of the MST is used twice (once when we go down the tree and once when we come back up)
    a. This results in a path that has a total weight of at most twice the weight of the MST, since every edge is counted twice
- Due to the triangle inequality, taking a direct path between any two nodes in the preorder traversal is no more expensive than the path through intermediate nodes in the MST
    a. This means that we can "shortcut" the traversal path without increasing the total weight, keeping it within twice the weight of the MST



Proof

Proof. The cost of a minimum spanning tree T is less than the cost of the optimal tour: $\text{cost}(T) \leq \text{cost}(A^*)$. Why?

A full walk W that "traces" the MST is of length $2\text{cost}(T)$ because every edge is crossed twice.

So: $\text{cost}(W) = 2\text{cost}(T) \leq 2\text{cost}(A^*)$.

W isn't a tour because it visits cities more than once. We can shortcut all but the *first* visit to a city. By the triangle inequality, this only reduces the cost of the tour.

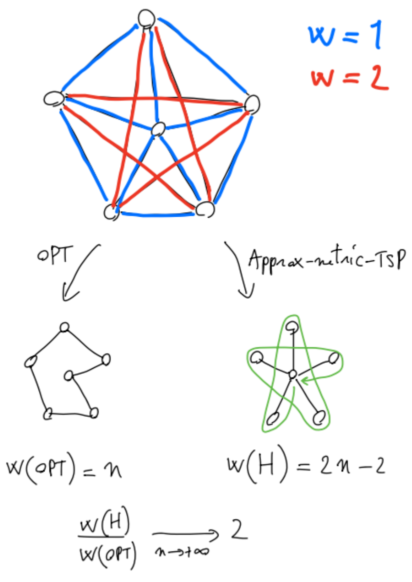
So: $\text{cost}(A) \leq 2\text{cost}(T) \leq 2\text{cost}(A^*)$. □

9.3.2 Exercises

Exercise

Show that the above analysis is tight by giving an example of a graph where $Approx\_Metric\_TSP$ returns a solution of cost $2 * H^*$.

Solution



Consider a complete graph of 6 vertices. We take the edges of weight 1 (blue) and the edges of weight 2. This satisfies the triangle inequality.

Here, OPT will use only edges of weight 1, as you can see from left graph, having as cost of the optimal tour n (all vertices with no cycle).

$Approx\_Metric\_TSP$ finds the minimum MST and doubles its edges to create a Hamiltonian cycle, resulting in a tour that visits each vertex twice, except for the central vertex.

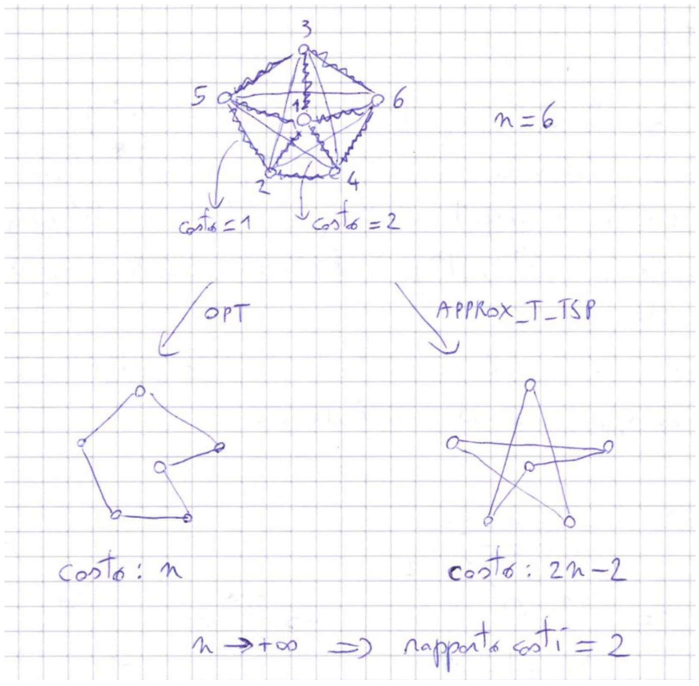
It multiplies the weight 2 over all vertices not considering the central one, so $2 * (n - 1) = 2n - 2$.

It's not hard to see that:

- The optimal solution uses only n edges of cost 1 → n
  a. (e.g. a complete graph with 6 edges with cost 1)
- The algorithm uses only edges of cost 2, apart from two edges of cost 1 → $2n - 2$

Over infinity, we have $\frac{2n-2}{n} = lim_{n\to\infty} \frac{2(n-1)}{n} = 2$

So, the formula is $\frac{Greedy}{OPT} \leq factor$.



Programming exercise

Implement *Approx_Metric_TSP* in whatever language you want and run it on TSPLIB– google the library (can be found here easily).

Solution

```
import math

from collections import defaultdict

def prim(graph, start):

    """

    Implements Prim's algorithm to find the minimum spanning tree (MST)
of a graph.

Args:

        graph (dict): A dictionary representing the graph, where keys are
vertices

        and values are dictionaries of adjacent vertices and their
weights.

        start (hashable): The starting vertex for Prim's algorithm.

    Returns:

        dict: A dictionary representing the MST, where keys are vertices
and

        values are tuples of the parent vertex and weight.
    """
    mst = {}
    visited = set()
    heap = [(0, start, None)]
    while heap:
        weight, u, parent = heappop(heap)
        if u in visited:
            continue
        visited.add(u)
        if parent is not None:
            mst[u] = (parent, weight)
        for neighbor, neighbor_weight in graph[u].items():
            if neighbor not in visited:
                heappush(heap, (neighbor_weight, neighbor, u))
    return mst
def preorder(tree, root):
    """
    Performs a preorder traversal of the given tree and returns the
ordered list of vertices.
    Args:
        tree (dict): A dictionary representing the tree, where keys are
vertices and
        values are tuples of the parent vertex and weight.
        root (hashable): The root vertex of the tree.
    Returns:
        list: A list of vertices in the order they were visited during
the preorder traversal.
    """
    visited = set()
    order = []
    def traverse(node):
        if node not in visited:
            visited.add(node)
            order.append(node)
            parent, _ = tree[node]
            if parent is not None:
                traverse(parent)
    traverse(root)
    return order
def approx_metric_tsp(graph):
    """
    Implements the Approx_Metric_TSP algorithm to find an approximate solution for the
    Traveling Salesman Problem (TSP) on a metric graph.
    Args:
        graph (dict): A dictionary representing the graph, where keys are vertices
        and values are dictionaries of adjacent vertices and their weights.
    Returns:
        list: A list of vertices representing the approximate TSP tour.
    """
    # Choose an arbitrary root vertex
    root = next(iter(graph))
    # Construct the minimum spanning tree (MST) using Prim's algorithm
    mst = prim(graph, root)
    # Perform a preorder traversal of the MST to get the ordered list of vertices
    ordered_vertices = preorder(mst, root)
    # Close the cycle by adding an edge from the last vertex to the root
    ordered_vertices.append(ordered_vertices[0])
    return ordered_vertices
```

This implementation includes the prim function to find the minimum spanning tree (MST) using Prim's algorithm, the preorder function to perform a preorder traversal of the MST, and the `approx_metric_tsp` function that combines these two functions to implement the `Approx_Metric_TSP` algorithm.

To run this implementation on the TSPLIB graphs, you'll need to download the TSPLIB dataset and parse the graph data into a format that can be used by the `approx_metric_tsp` function. Here's an example of how you can parse and run the algorithm on the `gr24.tsp` graph from the TSPLIB (said library offers many graphs, check above):

```
def parse_tsplib(file_path):
    """
    Parses a TSPLIB graph file and returns a dictionary representing the graph.
    Args:
        file_path (str): The path to the TSPLIB graph file.
    Returns:
        dict: A dictionary representing the graph, where keys are vertices
        and values are dictionaries of adjacent vertices and their weights.
    """
    graph = defaultdict(dict)
    with open(file_path, 'r') as file:
        lines = file.readlines()
        dimension = int(lines[3].split(':')[1])
        edges = lines[7 + dimension:]
        for edge in edges:
            u, v, weight = edge.split()
            u, v = int(u), int(v)
            graph[u][v] = int(weight)
            graph[v][u] = int(weight)
    return graph
# Parse the gr24.tsp graph
graph = parse_tsplib('gr24.tsp')
# Run the Approx_Metric_TSP algorithm
tour = approx_metric_tsp(graph)
print("Approximate TSP tour:", tour)
```

This code defines a `parse_tsplib` function to parse the TSPLIB graph file and convert it into a dictionary representation that can be used by the `approx_metric_tsp` function. It then parses the graph and runs the `approx_metric_tsp` algorithm on it, printing the approximate TSP tour.

Note that the TSPLIB dataset contains various instances of the TSP problem, and some of them may be too large to be efficiently solved by this implementation, which has a time complexity of O(n^2 log n) due to Prim's algorithm. For larger instances, you may need to use more efficient algorithms or optimizations.

9.4 3/2 (OR 1.5) APPROXIMATION ALGORITHM FOR METRIC TSP – CHRISTOFIDES

(Further readings: paper, article and article)

Christofides algorithm was born in 1976 and this will not be found in the CRLS, but inside the further readings free book here (page 46 to avoid wasting your/my time).

Reason for 2-approximation factor was the fact the preorder traversal of T^* used every edge of T^* exactly twice. We'll try to improve on this by constructing a tour that traverses MST edges only once.

The basic 2-approximation algorithm for Metric TSP involves using a Minimum Spanning Tree (MST) and a preorder traversal to create a tour. This approach, however, results in visiting each edge twice, leading to a tour that can be up to twice the weight of the optimal TSP tour.

Christofides' algorithm improves on this by ensuring that each edge is visited only once. To do so, we have to use an Eulerian graph. Here, a MST is created to make every node even, then shortcutting.

We give a couple of definitions useful for this context:

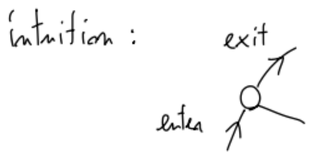
- A path (or cycle) is Eulerian if it crosses every edge of the graph exactly once
- A (connected) graph is Eulerian if there exists an Eulerian cycle

If the MST was Eulerian (cannot be) then we would have a 1-approx algorithm (which would be optimal, given one would cross every edge exactly once). $Approx\_Metric\_TSP$ is finding a "cheap" Eulerian cycle in the MST, but effectively needs to double its edges.

Question: is there a cheaper Eulerian cycle?

We quote the famous theorem by Euler, which was spawned by the Seven Bridges of Konigsberg mathematical problem, which, if you are curious, you can find here.

Theorem: A connected (multi)graph is Eulerian ⇔ every vertex has even degree. The intuition is the following: enter a vertex, then exiting from it using a new edge, doing that without using edges more than once.



We want to focus on the *odd* degree vertices, given I have to cross again vertices (the even ones are fine, given we don't pass on them again). Even better: if I want to traverse every edge exactly once, I must have a way out from every vertex.

Written by Gabriel R.

So, let's handle the odd-degree vertices of the MST explicitly.

Property: in any (finite) graph, the number of vertices of odd degree is even.

Proof: We use the following equality (*handshaking lemma* – the sum of the degrees [the numbers of times each vertex is touched] equals twice the number of edges in the graph)

$$\sum_{v\in V} \deg(v) = 2m$$

Basically, the sum of odd vertices with even ones, will get us an even result, that's the main intuition. So, we can split such summation into two parts:

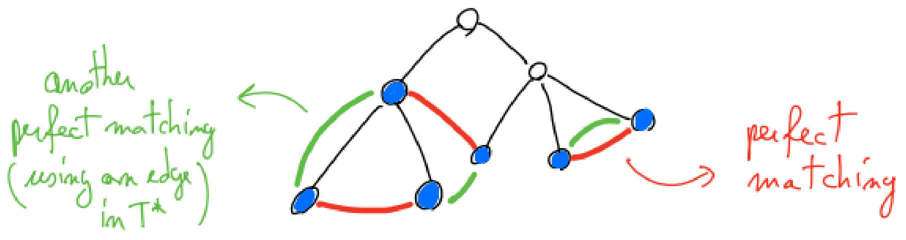
$$\underbrace{\sum_{u\in even} \deg(u)}_{even} + \underline{\sum_{w\in odd} \deg(w)} = \underbrace{2m}_{even}$$

Since the result must be *even*, the sum of degrees must be *even* too.

- This happens only if the number of odd degree vertices is *even*, and this happens since every edge covers 2 vertices

Idea: augment the initial MST T^* with (the cheapest basically) a minimum-weight perfect matching (perfect means that it includes all the vertices) between the vertices that have odd degree in the MST.

For instance, let's consider the following MST, coloring in blue the odd-degree vertices. Imagine we add a perfect matching colored in red, augmenting the previous MST T^* with a minimum-weight perfect matching in green, becoming as you can see from following figure.



⇒ the resulting graph has only even-degree vertices, i.e. is an Eulerian graph.

Let's write the algorithm, which does exactly *four* things:

$Christofides(G)$

1) $T^* \leftarrow Prim(G, r)$ $\quad // T^* = (V, E^*)$

2) Let D be the set of vertices of T^* with odd-degree (note $|D|$ is even). Compute a min-weight perfect matching M^* on the graph induced by D // this can be done in polynomial time (Edmonds, 1965)

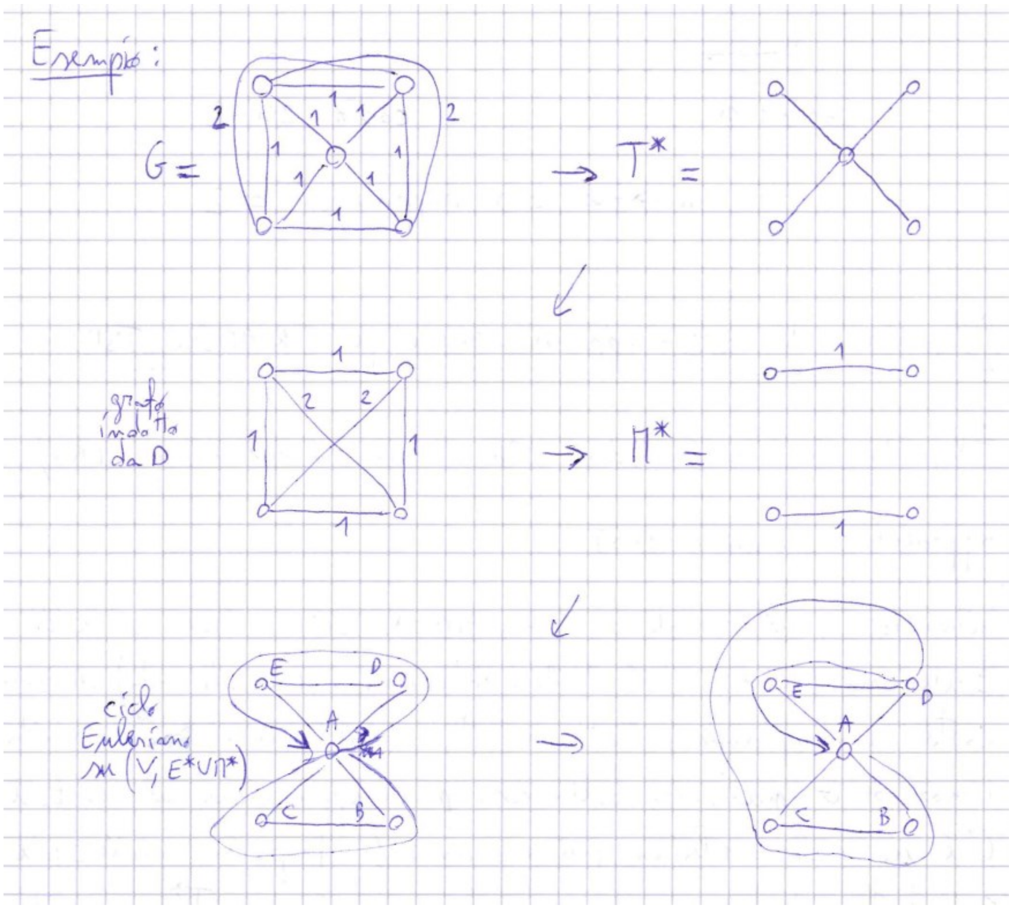
3) The graph $(V, E^* \cup M^*)$ is Eulerian // any edge in both E^* and M^* appears twice in this (multi)graph. Compute an Eulerian cycle E on this graph, called G^* so to have $G^* = (V, E^*) \cup M^*$.

4) Return the cycle that visits all the vertices of G in the order of their first appearance in the Eulerian cycle E (basically, skipping all repeated vertices – shortcutting)

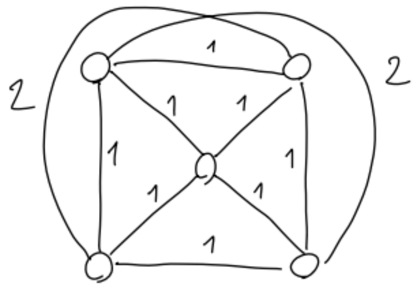
So, to summarize:

- The idea of $Approx\_Metric\_TSP$ is to approximate an optimal Metric TSP tour by using the Preorder traversal of the MST T^*
- As we have seen, this kind of traversal is like passing through every edge twice, but thanks to triangle inequality we are able to add shortcuts that do not increase the cost
- Christofides' algorithm, instead, exploits the concept of Eulerian cycle
- It creates an Eulerian graph starting from the MST T^*, and then it approximates an optimal Metric TSP tour by finding an Eulerian cycle in this new graph
- Finally, it adds shortcuts in order to make the Eulerian cycle (that can pass through the same vertex more the once) an Hamiltonian circuit

An interesting video on TSP and Christofides too here.

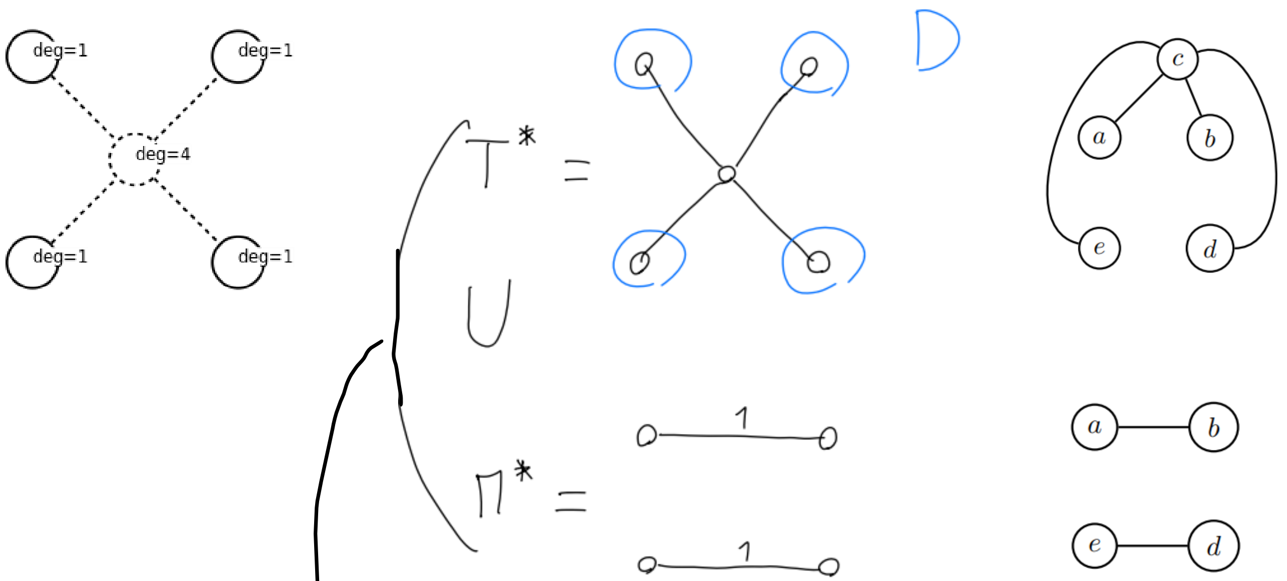


Consider the following example, connecting all vertices, in which edge weights obey the triangle inequality:

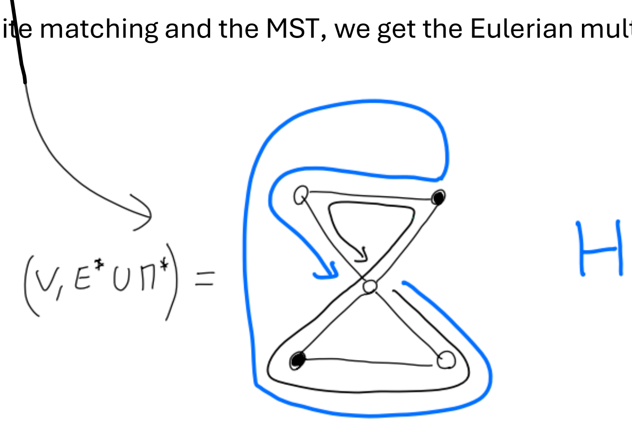


This set of figures appears in Wikipedia, in which there is the full explanation step by step. To keep coherence with our notes, this space is intentionally left blank to continue within the next page.

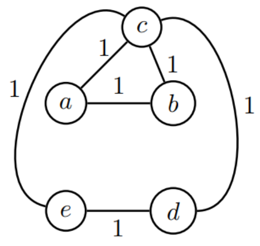
Now build the MST T^*, take the odd-degree vertices marked as blue in D and compute the minimum-weight perfect matching M^* (from the subgraph using only the odd-degree vertices).



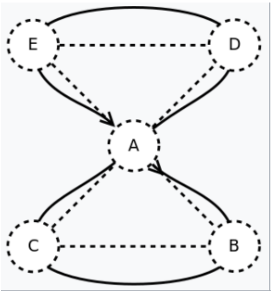
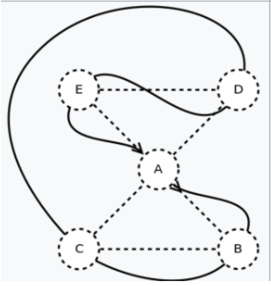
Merging together the unite matching and the MST, we get the Eulerian multigraph G^* as follows:



With this graph, Eulerian cycles can be computed, starting from a source vertex; consider, if we mark all the vertices for instance, the following result as G^*:



An Eulerian cycle computed on this graph can be: c, d, e, c, b, a, c. We can find shortcuts (basically, removing double occurrences of vertices apart from c, which is the source one – will be removed one since there are three) as we did for $Approx\_Metric\_TSP$ and the final Hamiltonian Tour H becomes: c, d, e, b, a, c.

	Calculate Euler tour Here the tour goes A->B->C->A->D->E->A. Equally valid is A->B->C->A->E->D->A.
	Remove repeated vertices, giving the algorithm's output. If the alternate tour would have been used, the shortcut would be going from C to E which results in a shorter route (A->B->C->E->D->A) if this is an euclidean graph as the route A->B->C->D->E->A has intersecting lines which is proven not to be the shortest route.

9.4.1 Analysis

- $w(H) \leq w(T^*) + w(M^*)$ - weight of the Hamiltonian cycle formed by shortcutting the Eulerian cycle is lower bound for optimal tour and the matching
    a. Holds by triangle inequality, given thanks to shortcutting, odd degrees in T^* were in even number, so H^* has an even number of vertices

- $w(T^*) \leq w(H^*)$ - weight of MST lower bound for optimal TSP tour
    a. Proven here already (last class w.r.t. this one)

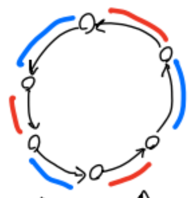
The goal to reach is $w(H) \leq \frac{3}{2} w(H^*)$. To do this, we would need to prove (can also be found here):

- $w(M^*) \leq^? \frac{1}{2} w(H^*)$ (by triangle inequality)

We will do the following clever step:

$w(\text{optimal tour of the odd-degree vertices of } T^*) \leq w(H^*)$

partition this in 2 perfect matchings:



even n° of vertices

One of these 2 has cost $\leq \frac{w(H^*)}{2}$ → optimal, since the perfect matching combines even degree vertices

Written by Gabriel R.

Putting all pieces together we get:

$$w(H) \le w(E^* \cup M^*) \le w(H^*) + \frac{w(H^*)}{2} = \frac{3}{2} w(H^*)$$

To conclude:

- Recent advancements: $(\frac{3}{2} - \epsilon)$-approx algorithm, with factor $e \sim 10^{-36}$
    a. See further reading on this one (also linked above or in 23/24 Moodle)
- Approx. ratio $\ge \frac{123}{122}$
- Conjecture: $\frac{4}{3}$

Not told in class, but to be complete, given the nature of this course: its worst case complexity is $O(n^3)$, given n the number of nodes or vertices of graph.

To conclude, two complete examples of Christofides runs:

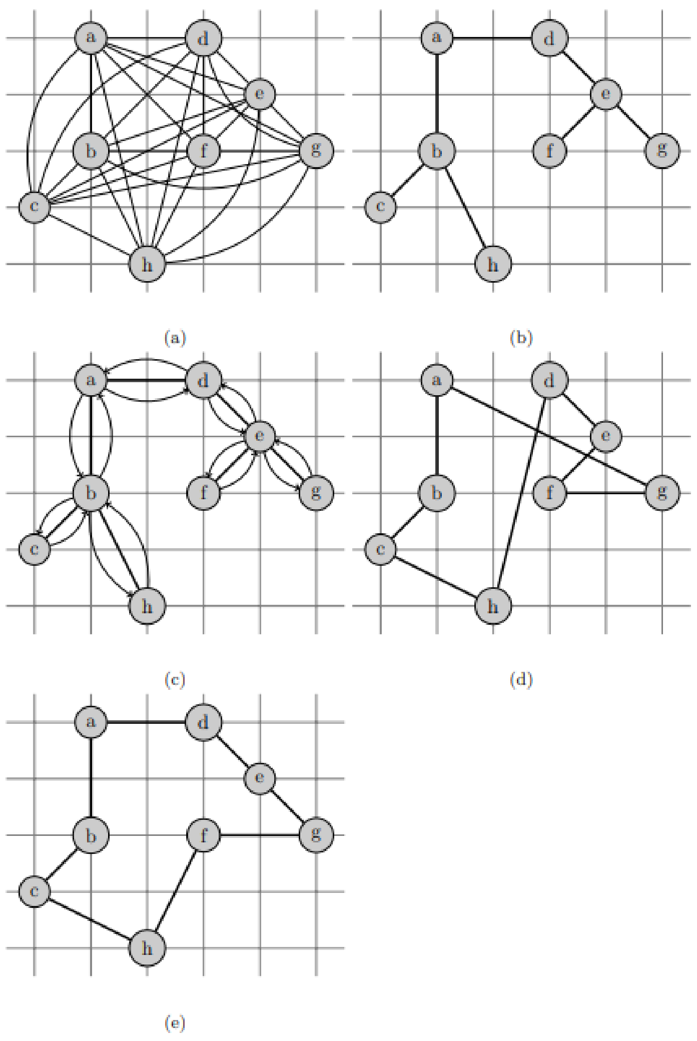
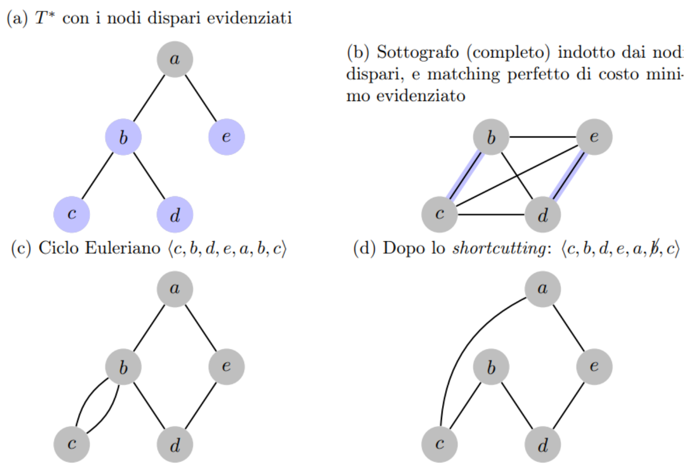


Figura 1.2: Esempio di applicazione dell'algoritmo di approssimazione per TSP

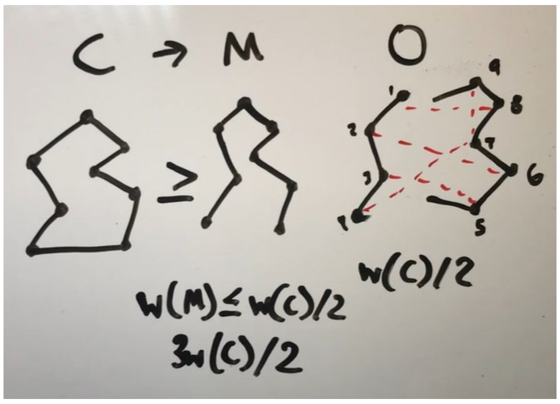
Figura 3.6: Algoritmo di Christofides


(a) T^* con i nodi dispari evidenziati
(b) Sottografo (completo) indotto dai nod dispari, e matching perfetto di costo minimo evidenziato
(c) Ciclo Euleriano $\langle c, b, d, e, a, b, c\rangle$
(d) Dopo lo *shortcutting*: $\langle c, b, d, e, a, \cancel{b}, c\rangle$

Il grafo ottenuto (prima dello *shortcutting*), è

$$G' = (V, E_{T^*} \cup M^*)$$

Another useful video on the topic (see minute 12):



On the approximation ratio, Wikipedia specifies the following (with some edits of mine I tried to put to better explain the steps of the problem):

- The cost of the solution produced by the algorithm is within 3/2 of the optimum. To prove this, let *C* be the optimal traveling salesman tour. Removing an edge from *C* produces a spanning tree, which must have weight at least that of the minimum spanning tree, implying that *w*(*T*) ≤ *w*(*C*) - lower bound to the cost of the optimal solution.

- The algorithm addresses the problem that T is not a tour by identifying all the odd degree vertices in T; since the sum of degrees in any graph is even (by the Handshaking lemma), there is an even number of such vertices. The algorithm finds a minimum-weight perfect matching M among the odd-degree ones.

- Next, number the vertices of *O* in cyclic order around *C*, and partition *C* into two sets of paths: the ones in which the first path vertex in cyclic order has an odd number and the ones in which the first path vertex has an even number.

    a. Each set of paths corresponds to a perfect matching of *O* that matches the two endpoints of each path, and the weight of this matching is at most equal to the weight of the paths

    b. Infact, each path endpoint will be connected to another endpoint, which also has an odd number of visits due to the nature of the tour

- Since these two sets of paths partition the edges of *C*, one of the two sets has at most half of the weight of *C*, and thanks to the triangle inequality its corresponding matching has weight that is also at most half the weight of *C*.

    a. The minimum-weight perfect matching can have no larger weight, so *w*(*M*) ≤ *w*(*C*)/2. Adding the weights of *T* and *M* gives the weight of the Euler tour, at most 3*w*(*C*)/2

    b. Thanks to the triangle inequality, even though the Euler tour might revisit vertices, shortcutting does not increase the weight, so the weight of the output is also at most 3*w*(*C*)/2

10 SET COVER

Set cover is an optimization problem that models many problems requiring resources to be allocated (basically, it's a generalization of Vertex Cover – or viceversa). It aims to find the *least* number of subsets that cover some universal set. A *cover* is a subcollection of sets which union covers X.

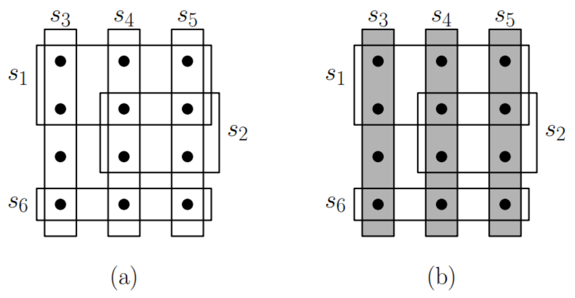


Fig. 1: Set cover. The optimum cover consists of the three sets $\{s_3, s_4, s_5\}$.

Its inputs are:

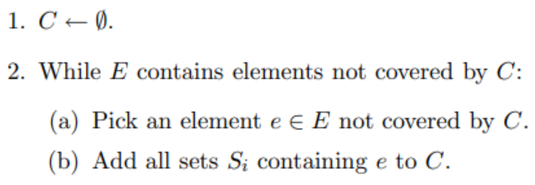
- $I = (X, F)$ = instance of the set covering problem
- $X =$ set of elements of any kind, called "universe"
- $F \subseteq \{S : S \subseteq X\} = B(X)$
    a. B stands for "Boolean" and F is the set of all subsets of X

There is a constraint that needs to be always respected: $\forall x \in X, \exists\, S \in F : x \in S$ i.e., "F covers X"

Optimization problem: (smallest subset of F having its members covering all X) → find $F' \subseteq F$ s.t.

1) F' covers X
2) $\min\ |F'|$

Example (also normal algorithm here on the right):



```
1. $C \leftarrow \emptyset$.
2. While $E$ contains elements not covered by $C$:
   (a) Pick an element $e \in E$ not covered by $C$.
   (b) Add all sets $S_i$ containing $e$ to $C$.
```

$X = \{1,2,3,4,5\}$

$F = \{\{1,2,3\}, \{2,4\}, \{3,4\}, \{4,5\}\}$

$\Rightarrow F^* = \{\{1,2,3\}, \{4,5\}\}$ (cover all elements using only two sets)

Applications:

- Hiring ($X =$ skills, $F =$ people with some skills)
    a. As a simple example, suppose that X represents a set of skills that are needed to solve a problem and that we have a given set of people available to work on the problem
    b. We wish to form a committee, containing as few people as possible, such that for every requisite skill in X, at least one member of the committee has that skill
- Spamming ($X =$ people, $F =$ mailing lists)

Assertion: Set Cover (in its decision version $\langle (X, F), k \rangle$) is NP-hard.

Proof: $Vertex\ Cover \leq_p Set\ Cover$

- Given an instance of Vertex Cover Problem $\langle G = (V, E), k \rangle$
- We create an instance of Set Cover problem $\langle (X, F), k \rangle$

Basically $\langle G = (V, E), k\rangle \rightarrow^f \langle (X, F), k\rangle$

where:

- $X = E$
- $F = \{S_1, S_2, \dots S_n\}$ one ∀ vertex ∈ V, $1,2, \dots n$
- $S_i = \{e = (u, v)$ such that $u = i$ or $v = i\}$, which is the set of covered vertices by edge e

Basically, there are $|V| = n$ subsets S_i, and each subset is the set of edges incident to vertex i.

Now show that finding a Set Cover of size k ⇔ finding a Vertex Cover of size k.

A bit more about the proof, just to make you understand completely the reduction working:

- (⇒) Suppose $\{S_1, S_2, \dots, S_k\}$ is a set cover for X. Then, every edge in E must be incident to at least one vertex $u_1, .., u_k$. This happens because every element is one node of the adjacency list and so we find the minimal number of nodes touching all edges of graph, guaranteeing it will be minimal (for all sizes, given, even if less than k). Therefore, it forms a vertex cover of size k in G.
- (⇐) Suppose $u_1, .., u_k$ is a vertex cover in G. Then, S_i covers all the edges incident to vertex u_i. Therefore, $\{S_1, .., S_k\}$ is a set cover of size k for X.

The transformation is linear in the size of the instance and preserves the approximation.

10.1 Greedy Approximation Algorithm

Think greedy, the professor says, make the simplest choice. The greedy method works by picking at each stage the set S that covers the greatest number of remaining elements that are uncovered:

- Choose the subset that contains the largest number of uncovered elements
- Remove from X those covered elements
- Repeat until $X = \emptyset$

```
$Approx\_Set\_Cover(X, F)$
    $U = X$
    $F' = \emptyset$ // solution (family being built)
    while $U \neq \emptyset$: do
        // take the set of F covering as many elements as possible
        let $S \in F = |S \cap U| = \max_{S' \in F}\{|S' \cap U|\}$
        $U \leftarrow U \setminus S$    // update the list of available elements, removing those from S
        $F \leftarrow F \setminus \{S\}$    // remove the sets already considered inside of F
        $F' \leftarrow F' \cup \{S\}$
    return $F'$
```

Correctness (aka "it does the job" – it covers all the elements): At every iteration $|U|$ decreases by at least one. In general, to prove the correctness of an approximation algorithm, it is sufficient to prove that the returned solution is always within the set of admissible solutions.

- Every element taken is always inside of the family of subsets containing such element
- Even when $U = \emptyset$, at every loop U decreases, so it can't even go out prematurely from while loop, given $\max_{S' \in F}\{|S' \cap U|\} \geq 1$ → every element is covered

Complexity:

- N. of iterations $\leq |X|$ (every $S_i \in F$ contains at least an element)
- N. of iterations $\leq |F|$ (every $S_i \in F$ contains at least two elements)
- ⇒ n. of iterations $\leq \min\{|X|, |F|\}$
- ∀ iterations the complexity is $\leq |X| * |F|$ (scanning all elements and decreasing elements in both sets)
- ⇒ $O(|X| * |F| * min\{|X|, |F|\})$
    a. It can be at most *cubic* in the *input size* (with the "right" data structure can be implemented efficiently in $O(|X| + |F|)$, i.e. in linear time)

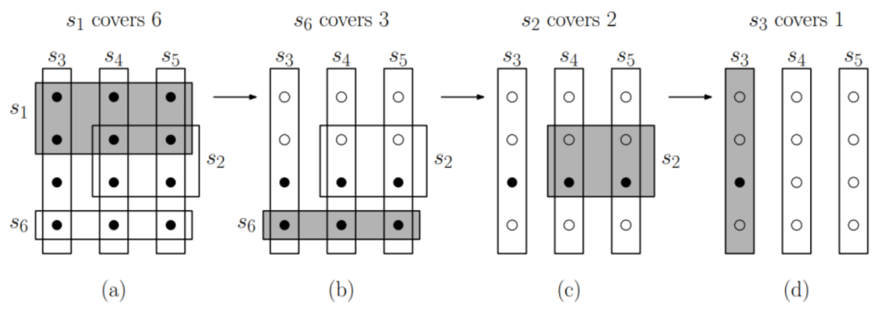


Fig. 2: The greedy heuristic. Final cover is $\{s_1, s_6, s_2, s_3\}$.

10.2 Analysis – A $\log_2(n)$ approximation algorithm

We'll show that $\frac{|F'|}{|F^*|} \leq \lceil \log_2(n) \rceil + 1$, where $n = |X|$ (we are showing $|F'| \leq f(|F^*|)$).

Property: if (X, F) admits a cover with $|F| \leq k$, then $\forall X' \subseteq X$ (X', F) admits a cover with $|F| \leq k$.

(This means that if I can cover X with k subsets, then I can definitely cover a subset of X with k subsets)

Idea: try to bound the number of iterations such that the set of <u>remaining elements</u> gets empty.

- $U_0 = X$
- $U_i =$ residual universe after then of the i^{th} iteration
- $|F^*| = k$ $\nearrow$ unknown (cardinality of optimal solution)

This is done limiting the number of loops to execute in such a way the set of elements gets empty as soon as possible.

Lemma: after the first k iterations, the residual universe is at least halved, that $|U_k| \leq \frac{n}{2}$

Being greedy, this can be seen as a recursive algorithm selecting a subset then repeating itself on the residual universe as follows:

$\Rightarrow$ after $k * i$ iterations $|U_{k-i}| \le \frac{n}{2^i}$ (after k iterations, the size of residual universe is the ones of remaining sets)

$\Rightarrow$ # (number) of necessary iterations $\lceil \log_2(n) \rceil * (k) + 1$ at each iteration $|F'| + +$

$\Rightarrow |F'| \le \lceil \log_2(n) \rceil * \underset{\|}{k} + 1$

$\Rightarrow |F'| \le \lceil \log_2(n) \rceil * |F^*| + 1$ (because at every iteration, $|F'|$ is increased by one)

Consider the "+1" here is present to cover the possible last element remaining to cover.

Let's prove the lemma in a proper way:

$\underset{\checkmark}{U_k \subseteq X} \Rightarrow U_k$ admits a cover size $\le k$ all in F (i.e. <u>not</u> yet selected by the algorithm)

(Trivial) *Property*: if (X, F) admits a cover with $|F| \le k$ then $\forall X' \subseteq X, (X', F)$ admits a cover with $|F| \le k$ (this happens because of the property above, given after k iterations, the residual universe has at most as many elements as the sets not yet selected)

Let $T_1, T_2, \dots, T_k \in F$ be those sets, where $\bigcup T_i$ covers U_k (covering all sets – residual universe after k iterations).

We apply the *pigeonhole* principle, which generally states that if n items are put into m containers, with $n > m$, then at least one container must contain more than one item.

- In other words and more precisely for the example and context here: given a set of elements where there is an order relation, there is always at least one element whose value is greater than the mean value

There are k subsets and there's the need to cover elements of cardinality U_k. It is possible and there is at least one which covers at least a fraction of all elements: $\exists \overline{T}$ s.t. $|U_k \cap \overline{T}| \ge \frac{|U_k|}{k}$ (pigeon hole)

We'll now see that in the first k iterations, $\forall$ iteration at least $\frac{|U_k|}{k}$ new elements get covered:

$\forall 1 \le i \le k$, let $S_i \in F$ be the selected subset of the algorithm. This subset has the following property:

$$|S_i \cap U_i| \ge |T_j \cap U_i| \; \forall 1 \le j \le k$$

This is true because at each interaction I, the cardinality of the intersection with the residual universe is at least as big as the cardinality of the T_j <u>not</u> selected (each interaction selects the set with biggest cardinality). This property is valid also for $\overline{T}$, that is:

$$|S_i \cap U_i| \ge |\overline{T} \cap U_i| \overset{|U_i| \ge |U_k|}{\ge} |\overline{T} \cap U_k| \ge \frac{|U_k|}{k}$$

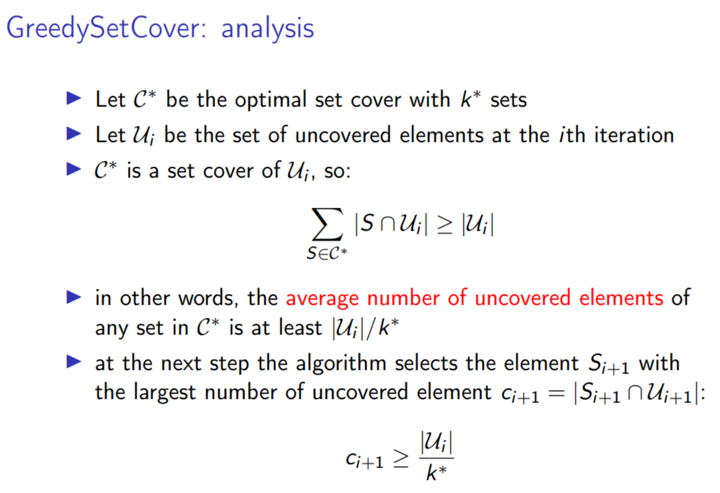
$\Rightarrow$ after the first k iterations the algorithm has covered $\frac{|U_k|}{k} * k = |U_k|$ elements

Since U_k is the set of elements not selected by the algorithm after k iterations, it follows that:

$$\underbrace{|U_k|}_{\text{residual}} \le n - \underbrace{|U_k|}_{\text{covered}}$$

satisfied for $|U_k| \leq \frac{n}{2}$ (after the k iterations, the residual universe is at least halved).

Taking screenshots from previous years to make the context even clearer:



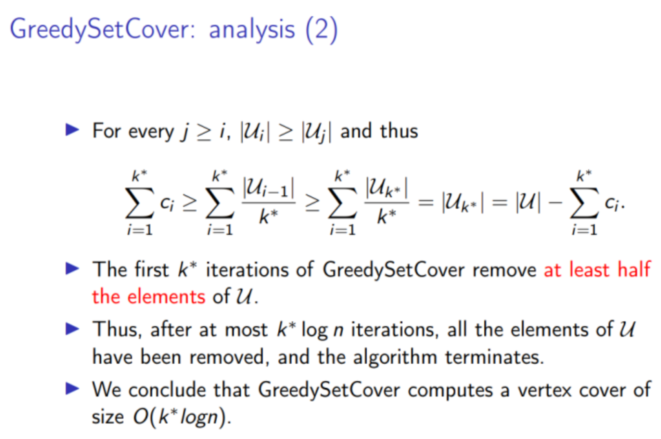
GreedySetCover: analysis

- Let $\mathcal{C}^*$ be the optimal set cover with k^* sets
- Let $\mathcal{U}_i$ be the set of uncovered elements at the ith iteration
- $\mathcal{C}^*$ is a set cover of $\mathcal{U}_i$, so:

$$\sum_{S \in \mathcal{C}^*} |S \cap \mathcal{U}_i| \geq |\mathcal{U}_i|$$

- in other words, the average number of uncovered elements of any set in $\mathcal{C}^*$ is at least $|\mathcal{U}_i|/k^*$
- at the next step the algorithm selects the element S_{i+1} with the largest number of uncovered element $c_{i+1} = |S_{i+1} \cap \mathcal{U}_{i+1}|$:

$$c_{i+1} \geq \frac{|\mathcal{U}_i|}{k^*}$$



GreedySetCover: analysis (2)

- For every $j \geq i$, $|\mathcal{U}_i| \geq |\mathcal{U}_j|$ and thus

$$\sum_{i=1}^{k^*} c_i \geq \sum_{i=1}^{k^*} \frac{|\mathcal{U}_{i-1}|}{k^*} \geq \sum_{i=1}^{k^*} \frac{|\mathcal{U}_{k^*}|}{k^*} = |\mathcal{U}_{k^*}| = |\mathcal{U}| - \sum_{i=1}^{k^*} c_i.$$

- The first k^* iterations of GreedySetCover remove at least half the elements of $\mathcal{U}$.
- Thus, after at most $k^* \log n$ iterations, all the elements of $\mathcal{U}$ have been removed, and the algorithm terminates.
- We conclude that GreedySetCover computes a vertex cover of size $O(k^* logn)$.

10.2.1 Exercises

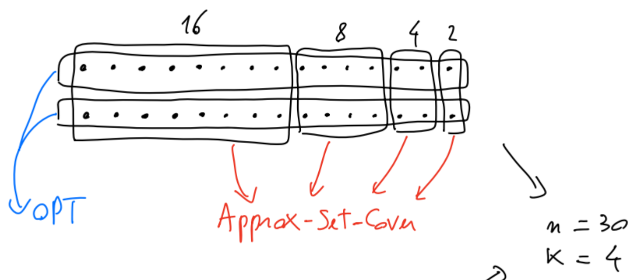
Is this analysis tight?

Exercise: *show that there is an input $I = (X, F)$ on which Approx_Set_Cover achieves an approximation ratio of $\theta(\log(n))$*

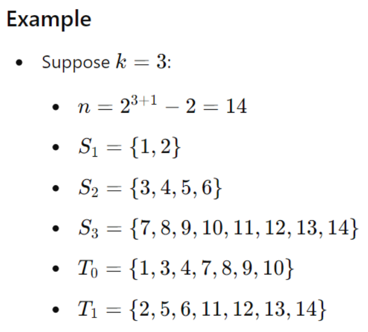
(Hint: the algorithm chooses the set that contains the largest n. of uncovered elements, whereas OPT chooses a set that contains the second largest n. of uncovered elements)

Solution

Consider the following schema, applying exactly what the hint told – we have 30 elements, in which the optimal choice would be to select both the complete sets of elements, while the algorithm selects progressively only a fraction of those:



- X has $n = 2^{k+1} - 2$ elements for some $k \in N$
- F has:
  a. k pairwise disjoint sets $S_1, \dots S_k$ with sizes $2, 4, \dots, 2^k$
  b. Two additional disjoint sets T_0, T_1
     i. Each of which contains half of the elements from each S_i



Example

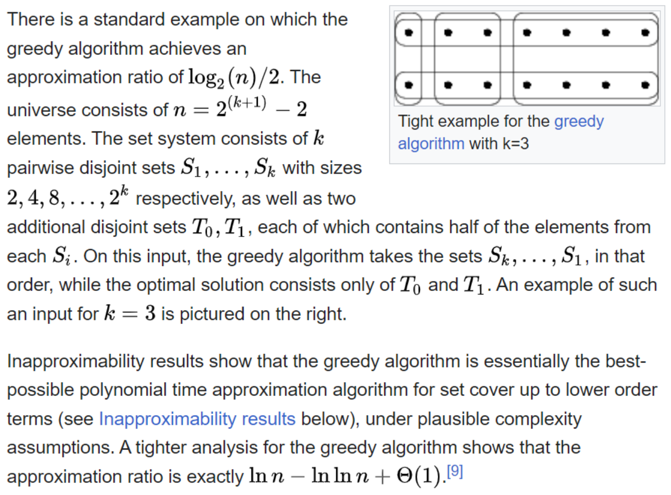
- Suppose $k = 3$:
  - $n = 2^{3+1} - 2 = 14$
  - $S_1 = \{1, 2\}$
  - $S_2 = \{3, 4, 5, 6\}$
  - $S_3 = \{7, 8, 9, 10, 11, 12, 13, 14\}$
  - $T_0 = \{1, 3, 4, 7, 8, 9, 10\}$
  - $T_1 = \{2, 5, 6, 11, 12, 13, 14\}$

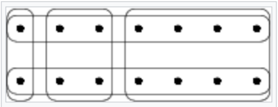
Written by Gabriel R.

So, in the end:

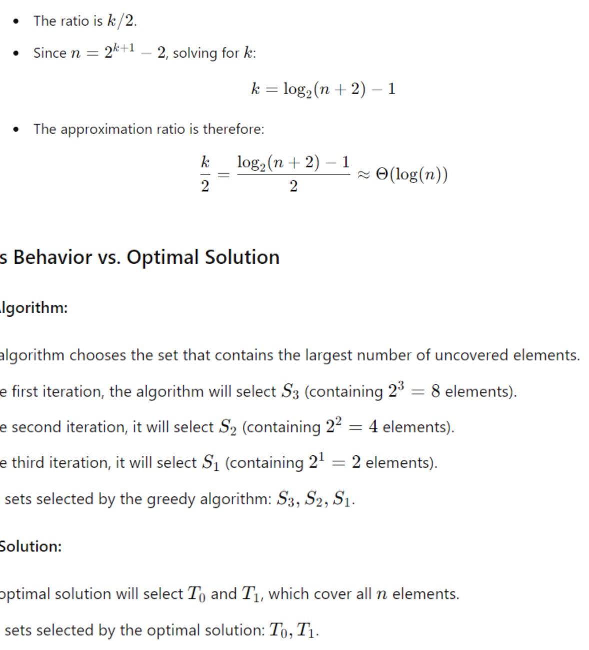
- $Approx\_Set\_Cover \rightarrow S_k, S_{k-1}, \dots S_1$
- $OPT \rightarrow T_0, T_1$
- ratio: $\frac{k}{2} = \theta(\log(n))$

This is also visible (slightly different) from Wikipedia here:


There is a standard example on which the greedy algorithm achieves an approximation ratio of $\log_2(n)/2$. The universe consists of $n = 2^{(k+1)} - 2$ elements. The set system consists of k pairwise disjoint sets $S_1, \dots, S_k$ with sizes $2, 4, 8, \dots, 2^k$ respectively, as well as two additional disjoint sets T_0, T_1, each of which contains half of the elements from each S_i. On this input, the greedy algorithm takes the sets $S_k, \dots, S_1$, in that order, while the optimal solution consists only of T_0 and T_1. An example of such an input for $k = 3$ is pictured on the right.


Tight example for the greedy algorithm with k=3

Inapproximability results show that the greedy algorithm is essentially the best-possible polynomial time approximation algorithm for set cover up to lower order terms (see Inapproximability results below), under plausible complexity assumptions. A tighter analysis for the greedy algorithm shows that the approximation ratio is exactly $\ln n - \ln \ln n + \Theta(1)$.[9]


- The ratio is $k/2$.
- Since $n = 2^{k+1} - 2$, solving for k:

$$k = \log_2(n+2) - 1$$

- The approximation ratio is therefore:

$$\frac{k}{2} = \frac{\log_2(n+2) - 1}{2} \approx \Theta(\log(n))$$

Algorithm's Behavior vs. Optimal Solution

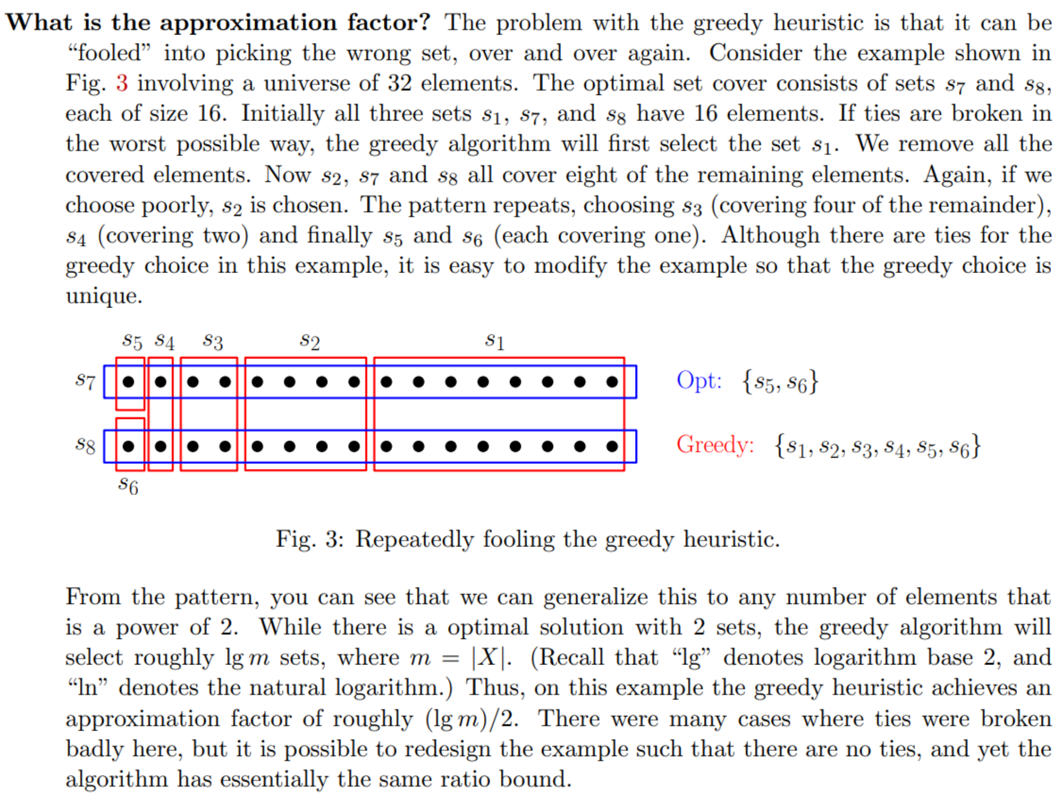
1. **Greedy Algorithm:**
   - The algorithm chooses the set that contains the largest number of uncovered elements.
   - In the first iteration, the algorithm will select S_3 (containing $2^3 = 8$ elements).
   - In the second iteration, it will select S_2 (containing $2^2 = 4$ elements).
   - In the third iteration, it will select S_1 (containing $2^1 = 2$ elements).
   - Total sets selected by the greedy algorithm: S_3, S_2, S_1.
2. **Optimal Solution:**
   - The optimal solution will select T_0 and T_1, which cover all n elements.
   - Total sets selected by the optimal solution: T_0, T_1.

So, basically:

- The greedy algorithm selects k sets
- The optimal selects 2 sets

No approximation is currently better than $\log(n)$, unless $P \neq NP$.

An even better explanation of this example coming from here:


What is the approximation factor? The problem with the greedy heuristic is that it can be "fooled" into picking the wrong set, over and over again. Consider the example shown in Fig. 3 involving a universe of 32 elements. The optimal set cover consists of sets s_7 and s_8, each of size 16. Initially all three sets s_1, s_7, and s_8 have 16 elements. If ties are broken in the worst possible way, the greedy algorithm will first select the set s_1. We remove all the covered elements. Now s_2, s_7 and s_8 all cover eight of the remaining elements. Again, if we choose poorly, s_2 is chosen. The pattern repeats, choosing s_3 (covering four of the remainder), s_4 (covering two) and finally s_5 and s_6 (each covering one). Although there are ties for the greedy choice in this example, it is easy to modify the example so that the greedy choice is unique.

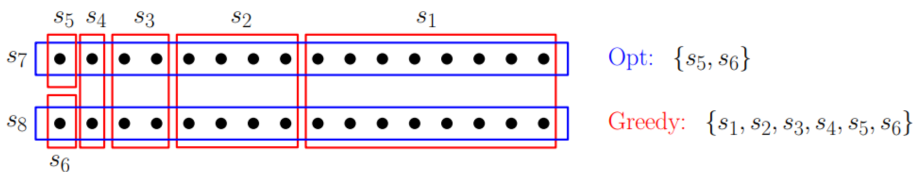


Fig. 3: Repeatedly fooling the greedy heuristic.

From the pattern, you can see that we can generalize this to any number of elements that is a power of 2. While there is a optimal solution with 2 sets, the greedy algorithm will select roughly $\lg m$ sets, where $m = |X|$. (Recall that "lg" denotes logarithm base 2, and "ln" denotes the natural logarithm.) Thus, on this example the greedy heuristic achieves an approximation factor of roughly $(\lg m)/2$. There were many cases where ties were broken badly here, but it is possible to redesign the example such that there are no ties, and yet the algorithm has essentially the same ratio bound.

11 Randomized Algorithms

(Further readings: here)

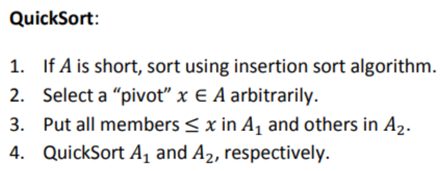
11.1 Overview

Randomized algorithms are algorithms that may do *random* choices, basically using a source of randomness in its logic. Why? It may seem paradoxical, but adding these kinds of choices can make them simpler and faster. We would give some examples here:

- Example 1: Randomized quicksort (RQS) – code on the side to make you remember QS

The quicksort algorithm performs multiple steps. Each step chooses a pivot element and places it in its correct location. The random shuffle approach chooses the pivot location at random, basically "breaking" the bad input instance. If every recursive call does a random choice, it's very hard to take always the "unlucky" element (which happens if the pivot is chosen progressively)

pivot
n elements


QuickSort:

1. If A is short, sort using insertion sort algorithm.
2. Select a "pivot" $x \in A$ arbitrarily.
3. Put all members $\leq x$ in A_1 and others in A_2.
4. QuickSort A_1 and A_2, respectively.

In the worst case, quicksort has a complexity of $T_{QS}(n) = O(n^2)$.

Randomized quicksort (RQS) chooses the pivot at random. The expected complexity of RQS is: $E[T_{RQS}] = O(n \log(n))$ (here E = expectation, which is the "expected value" in probability, something that on average happens).

- Choosing an element at random helps finding "more or less" an element in the middle of the sequence
- This choice at random hides the worst-case inputs
    a. Not knowing the algorithm's moves in advance

We will come back to this algorithms at the end of this file, as one as of the last lessons.

- Example 2: verifying polynomial identity
    a. (more discussion on this in "Further readings" of Moodle or above)

The name says it all: check whether two polynomials are equivalent. Consider the following, where one asks if the polynomial not in normal form and the other in canonical form:

$$\underbrace{(x+1)(x-2)(x+3)(x-4)(x+5)(x-6)}_{H(x)} \overset{?}{\equiv} \underbrace{x^6 - 7x^3 + 25}_{G(x)}$$

The obvious algorithm would transform $H(x)$ in canonical form $\sum_{i=0}^{d=6} c_i x^i$ and verify whether all the coefficients c_i of all monomials are equal.

The algorithm is pretty slow; considering d =maximum degree, the complexity would be $O(d^2)$.

Consider now a faster algorithm:

```
- Choose a random integer $r$     // $new\ operation$
- Compute $H(r)$                  // $O(d)$
- Compute $G(r)$                  // $O(d)$
- If $H(r) = G(r)$
    a. then return YES
    b. else return NO
```

This algorithm is linear, but does this work? Consider the following example:

$$r = 2$$

$$H(2) = 0 \qquad \Rightarrow H(x) \neq G(x)$$

$$G(2) = 33$$

In this case the algorithm always outputs the correct answer, but what if $H(r) = G(r)$?

Consider the following example:

$$x^2 + 7x + 1 \equiv^? (x+2)^2$$

$$r = 2:\ 19 \neq 16$$

$$r = 1:\ 9 = 9$$

unlucky choice of r

The algorithm returns YES even if the two polynomial are not equivalent, so the choice is wrong.

- If the equations is correct, the algorithm is always correct
- Otherwise, the algorithm returns the wrong answer only if r is a root of the polynomial
$$F(x) = G(x) - H(x) = 0$$

So basically, if the algorithm outputs NO is always correct, but may fail when it outputs YES. Therefore, the algorithm fails only if it is unlucky in choosing the value of r. But how likely is to choose such a value of r?

If $r \in \{1,2,\dots,100d\}$ where d is the max degree of $F(x)$ then:

$$\Pr(algorithm\ fails) \leq \frac{d}{100d} = \frac{1}{100}$$

small, but still not satisfactory

where "algorithm fails" means it returns how much the algorithm is wrong.

- It's unlikely that the algorithm fails
    a. But still not satisfactory, given there is a 1% error
- What if I can make this probability much lower than that?

So we ask: how to reduce the possibility of error?

- Run the algorithm 10 times (run multiple times and decrease exponentially the error)
- If YES in all the 10 runs
    a. then return YES
    b. else return NO

Now the probability of error is much lower, given the algorithms fails only if it fails all the 10 times. Probabilities are independent from each other, so you have to multiply them all:

$$\Pr(algorithm\ fails) \leq \left(\frac{1}{100}\right)^{10} = 10^{-20} < 2^{-64}$$

2^{-64} is comparable to the probability of a hardware error in your computer by cosmic radiation (quoting Donald Knuth, one of the most famous Computer Scientists). So, this algorithm is correct for all practical purposes.

11.2 Classification of Randomized Algorithms

We divide these into two main categories:

1) Randomized algorithms that <u>never fail</u>, which are called "<u>LAS VEGAS</u>" algorithms
    a. (E.g., randomized quicksort)

$$\forall i \in I, A_R(i) = \overset{\text{a solution}}{s} \quad s.t. (i, s) \in \Pi$$

where $\Pi \subseteq I\ x\ S$ is the decisional problem, i is an input instance, A_R is the random algorithm which applied to the input instance produces a solution s s.t. the couple (i, s) belongs to Π

Observation: s may not be the same $\forall i$.

Randomness comes into play in the analysis of the complexity – because it depends from the randomness of the choices. $\forall n, T(n)$ is a <u>random variable</u> of which we usually study its expectation $E[T(n)]$ or $\Pr\big(T(n) > c * f(n)\big) \to \leq \frac{1}{n^k}$ (so, for some constants c and k, we say that $T(n) = O(f(n))$ <u>with high probability</u> (here, $T(n)$ is called complexity function) – this second one is more powerful than the first, so Pr more powerful than E → $\Pr(A_\Pi(i)\ terminates \geq c * f(n)\ steps) \leq \frac{1}{n^d}$

- This is done in order to reduce the probability the algorithm is not precise, hence the probability the algorithm takes "X" time is very low
- Here the input is assumed to be from a probability distribution

The space of probabilities corresponds to the random choices made by the algorithm.

- (Do not confuse this with the probabilistic analysis of a deterministic algorithm, where the space of probabilities = distribution of the inputs)

Specifically, the *expected* runtime be finite, where the expectation is carried out over the space of random information, or entropy, used in the algorithm

2) Randomized algorithms that may fail are called "MONTE CARLO" algorithms
    a. E.g., verifying polynomial identities

$$\forall i \in I, A_R(i) = s \;\; s.t. \;\; (i,s) \notin \Pi$$

We study $\Pr((i,s) \notin \Pi)$ as a function of $n = |i|$ → family of random variables (binary)

Moreover, even $T(n)$ may be a random variable. For decision problems, these algorithms can be divided into:

- One-sided: they may fail only on *one* answer
    a. E.g., can make right all YES instances but may be wrong on all NO instances
- Two-sided: they may fail in *both* answers
    a. E.g., it can make wrong all YES instances but can make wrong all NO instances

In this course, we will see:

- One LAS VEGAS algorithm
    a. Randomized Quicksort – we'll see a high probability analysis
- One MONTE CARLO algorithm
    a. Karger's algorithm for Minimum Cut – again, with an analysis in high probability

Definition: given $\Pi \subseteq I \times S$ an algorithm A_Π has complexity $T(n) = O\big(f(n)\big)$ with high probability (w.h.p) if ∃ constants $c, d > 0$ s.t. $\forall i \in I, |i| = n$,

$$\Pr\,(A_\Pi(i)\; terminates\; in > c * f(n)\; steps) \le \frac{1}{n^d}$$

$$\rightarrow O(f(n))\; w.p. > 1 - \frac{1}{n^d} \rightarrow_{n \to +\infty} 1$$

Definition: given $\Pi \subseteq I \times S$ an algorithm A_Π is correct with high probability (w.h.p) if ∃ constant $d > 0$ s.t. $\forall i \in I, |i| = n, \Pr\,((i, A_\Pi) \notin \Pi \le \frac{1}{n^d}$

The characterization with high probability is always more powerful that the average case.

$high\; probability \Rightarrow expected$ (viceversa usually is not true)

11.2.1 Markov's lemma

Exercise

Assume (by hypothesis) that:

1) *A_Π LAS VEGAS, with $T_{A_\Pi}(n) = O(f(n))$ w.h.p; in particular,* $\Pr\Big(T_{A_\Pi}(n) > c * f(n)\Big) \le \frac{1}{n^d}$
2) *A_Π has a worst-case deterministic complexity $O(n^a), a \le d\; \forall n$*

Show that $E\big[T_{A_\Pi}(n)\big] = O(f(n))$ (that means, in high probability we get the average value, usually not true).

We will apply the following:

Markov's lemma: let T be a non-negative, bounded ($= b \in \mathbb{N}\ s.t. \Pr(T > b) = 0$), integer, random variable. Then $\forall t$ s.t. $0 \le t \le b$,

$$\underline{t * \Pr(T \ge t)} \le E[T] \le t + (b - t)\Pr\,(T \ge t)$$

This basically gives an upper bound on the probability that a non-negative random variable is greater than or equal to some positive constant (usually you see the first inequality).

Proof

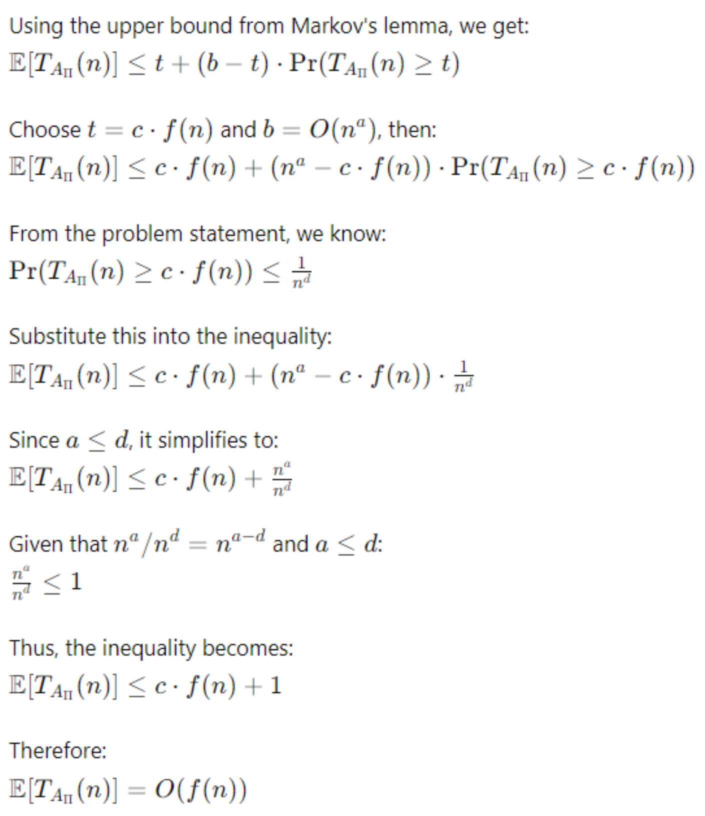
Using the upper bound of the lemma, the running time $T_{A_\Pi}(n)$ is non-negative and bounded by $O(n^a)$. Hence, $b = O(n^a)$. For the bound, we have $c * f(n) \le t \le O(n^a)$.

$$E[T_{A_\Pi}(n)] \le \underbrace{c * f(n)}_{t} + \frac{\underbrace{(n^a - c)}_{b-t} * f(n)}{\underbrace{n^d}_{Pr(T \geqslant t)}}$$

$$\le c * f(n) + \frac{n^a}{n^d} \le c * f(n) + 1$$

$$= O(f(n))$$

A more complete explanation here:



Using the upper bound from Markov's lemma, we get:
$\mathbb{E}[T_{A_\Pi}(n)] \le t + (b - t) \cdot \Pr(T_{A_\Pi}(n) \ge t)$

Choose $t = c \cdot f(n)$ and $b = O(n^a)$, then:
$\mathbb{E}[T_{A_\Pi}(n)] \le c \cdot f(n) + (n^a - c \cdot f(n)) \cdot \Pr(T_{A_\Pi}(n) \ge c \cdot f(n))$

From the problem statement, we know:
$\Pr(T_{A_\Pi}(n) \ge c \cdot f(n)) \le \frac{1}{n^d}$

Substitute this into the inequality:
$\mathbb{E}[T_{A_\Pi}(n)] \le c \cdot f(n) + (n^a - c \cdot f(n)) \cdot \frac{1}{n^d}$

Since $a \le d$, it simplifies to:
$\mathbb{E}[T_{A_\Pi}(n)] \le c \cdot f(n) + \frac{n^a}{n^d}$

Given that $n^a/n^d = n^{a-d}$ and $a \le d$:
$\frac{n^a}{n^d} \le 1$

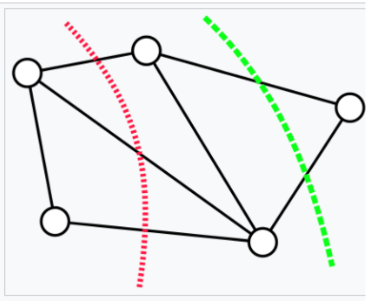
Thus, the inequality becomes:
$\mathbb{E}[T_{A_\Pi}(n)] \le c \cdot f(n) + 1$

Therefore:
$\mathbb{E}[T_{A_\Pi}(n)] = O(f(n))$

11.3 Karger's Algorithm for Minimum Cut

A quite simple MONTE CARLO and elegant algorithm created in 1993. Let's start from the problem itself it wants to solve: the minimum cut revolves finding a cut of minimum size, that is, the minimum number of edges whose removal disconnects the graph.

Below, you can see an example to help yourself:

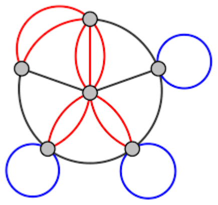


A graph and two of its cuts. The dotted line in red represents a cut with three crossing edges. The dashed line in green represents one of the minimum cuts of this graph, crossing only two edges.[1]

Applications: network reliability. war, computer graphics, etc.

Remark: we are talking about unweighted graphs, but it's also studied in weighted graphs. As a personal side note, this is the dual problem of maximum flow. As of now, there are no deterministic algorithms doing better.

Karger actually solves a more general problem: minimum cut on multigraphs (i.e., multiple edges between two vertices are allowed) – an example of one in the right figure.



Definition: A multiset is a collection of objects with repetitions allowed. It's usually denoted between a couple of brackets, as you can see here.

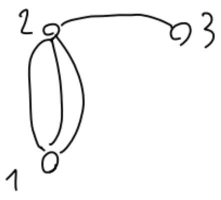
$$S = \{\{v : v \in S - objects\}\}$$

$$\forall\ object\ o \in S, m(o) \in \mathbb{N} \setminus \{0\}$$

where m = multiplicity, so how many copies of o are in S.

Definition: a multigraph $G = (V, \mathcal{E})$ s.t. $\forall\ V \subseteq \mathbb{N}$, V finite and $\mathcal{E}$ is a multiset of elements $(u, v)\ s.t.\ u \neq v$ (considering $\mathcal{E}$ is not an uppercase epsilon, but in LaTeX is written as (\mathcal(E)) and it's pronounced normally "E").

The following is the one seen in course example:



Note: A simple graph $G = (V, E)$ is also a multigraph.

Definition: given $G = (V, \mathcal{E})$ connected, a <u>cut</u> $C \subseteq \mathcal{E}$ is a multiset of edges s.t. $G' = (V, \mathcal{E} \setminus C)$ is not connected.

- A *cut* is a set of edges disconnecting the starting graph
    a. Creating at least two connected components
- Consider the following notions:
    a. *Path* in a multigraph = sequence of vertices where ∀ couple of vertices ∃ an edge
    b. *Connectivity* in a multigraph = a multigraph is connected if ∀ couple of vertices ∃ a path connecting them
- The problem is to choose the *smallest* set of edges that *disconnects* the multigraph

Actually, the probability that at the first run Karger's algorithm returns a minimum cut is not high. The idea is to repeat the procedure k times to reduce the probability of error. The value of k will be determined by the analysis of the algorithm.

Let's give Karger's idea here:

1. Choose an edge at random

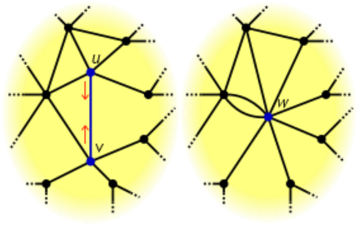
2. "Contract" the two vertices of that edge, removing all the edges incident both vertices

3. Repeat until only 2 vertices remain: return the edges between those vertices

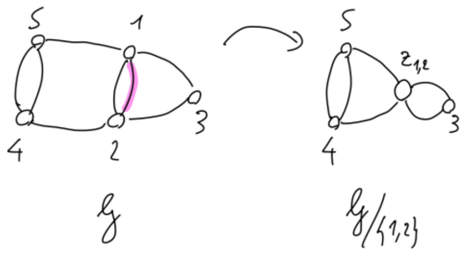
This works with very low probability, but let's use the trick we saw already: repeating this a good enough number of times, can actually refine the analysis and obtain a good level of probability.

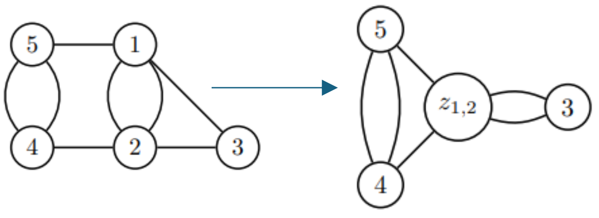
We see below two examples of the same contraction in Karger and on the right a generic contraction.

- Basically, it makes the two vertices to collapse in just one vertex connected with all the previous adjacent vertices
- If as a result there are several edges between some pairs of (newly formed) vertices, retain them all



example :





Definition: given $G = (V, \mathcal{E})$ and $e = (u, v) \in \mathcal{E}$, the <u>contraction of G with respect to e</u>, $\frac{G}{e} = (V', \mathcal{E}')$ is the multigraph with $V' = V \setminus \{u, v\} \cup \{z_{u,v}\}$ with $z_{u,v} \notin V$ coming from the fusion of u and v:

$$\mathcal{E}' = \mathcal{E} \setminus \{\{(x, y)\ s.t.\ (x = u)\ or\ (x = v)\}\}$$

$$\cup \{\{\left(z_{u,v}, y\right)\ s.t.\ (u, y) \in \mathcal{E}\ or\ (u, y) \in \mathcal{E}, y \neq u\ and\ y \neq v$$

It follows that:

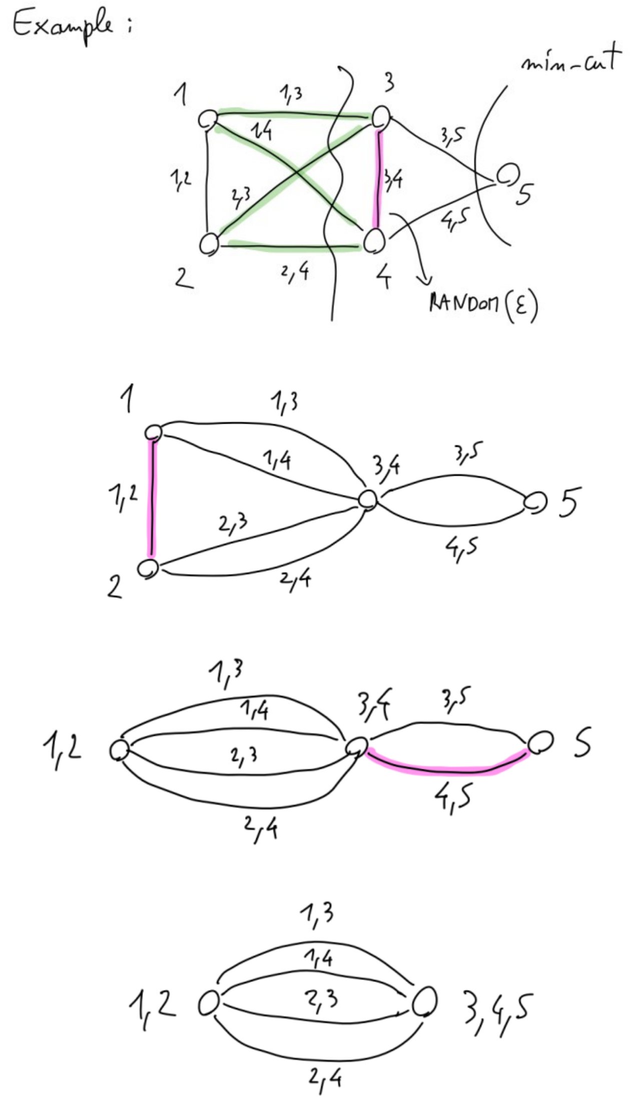
- $|V'| = |V| - 1 \Rightarrow n - 2$ iterations are needed $\qquad (= |V| - 2 + 1)$
- $|\mathcal{E}'| = |\mathcal{E}| - m(e) \leq |\mathcal{E}| - 1$ (the edge linking two nodes to contract disappears)

Written by Gabriel R.

We describe the algorithm here:

```
$FULL\_CONTRACTION\ (G = (V, \mathcal{E}))$
    for $i = 1$ to $n - 2$: do
        $e \leftarrow RANDOM(\mathcal{E})$                  $//\ choose\ an\ edge\ at\ random\ in\ multiset$
        $G' = (V', \mathcal{E}') \leftarrow \frac{G}{e}$    $//\ contraction\ on\ graph\ with\ random\ edge\ selection$
        $V \leftarrow V'$                                   $//\ construct\ the\ new\ graph\ with\ vertices\ and\ edges$
        $\mathcal{E} \leftarrow \mathcal{E}'$
    return $|\mathcal{E}|$                                  $//\ return\ the\ graph\ and\ its\ cardinality$
```

Let's see here an example of the algorithm running, in which we contract vertices step by step choosing edges at random, not choosing the correct min-cut as you can see:



⇒ <u>unsuccessful</u> run of $FULL - CONTRACTION$, because it didn't choose the correct minimum cut upfront as said, so we end up with more edges than the ones actually needed in contraction. This happens because the algorithm is randomized, so finding the minimum cut in the first run is just luck: that's why we have to do multiple runs to make higher the probability of a good run.

Written by Gabriel R.

Running the algorithm for $k = \binom{n}{2}$ times, the probability of success can be made arbitrarily high.

The following example can make your ideas clearer on this (I hope):

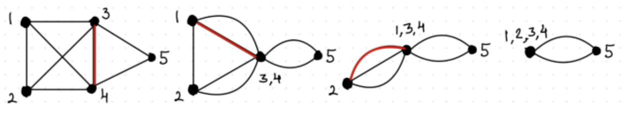


Figure 1.1: A successful run of Karger's min-cut algorithm

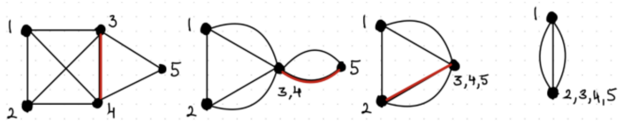


Figure 1.2: An unsuccessful run of Karger's min-cut algorithm

We give here the full Karger algorithm now, which idea is "hoping" to do a contraction on an edge of the minimum cut so to preserve said minimum cut, which is then returned as you can see.

Consider k → how many times to repeat $FULL\_CONTRACTION$, which depends on the probability of making a mistake – hence, it depends on the analysis of the algorithm)

```
$KARGER(G = (V, \mathcal{E}), k)$
    $min - cut = \mathcal{E}$
    for $i = 1$ $to$ $k$: do
        $t = FULL\_CONTRACTION(G)$
        if $|t| < |min - cut|$ then
            $min - cut = t$
    return $min - cut$
```

→ repeats $FULL\_CONTRACTION$ k times to reduce the probability of error

to be determined by the analysis

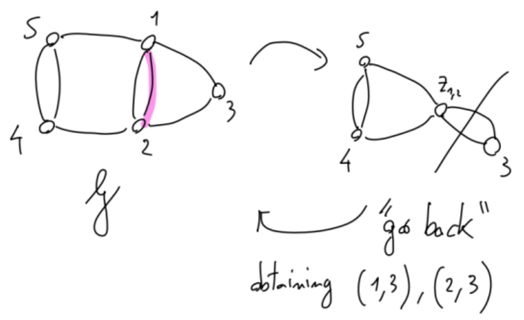
11.4 Analysis of Karger's Algorithm

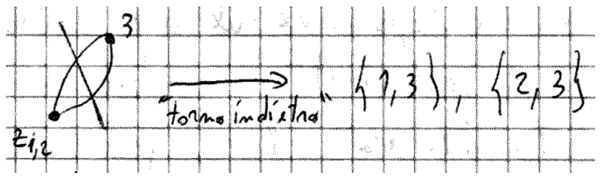
We'll show for which value of k the algorithm returns a minimum cut with high probability, effectively proving the cardinality of the min-cut does not decrease.

Property: $\forall$ cut C' in $\frac{G}{e}$ $\exists$ a cut C in G of the same cardinality $\Rightarrow \left|min - cut\ in \frac{G}{e}\right| \geq |min - cut\ in\ G|$.

Proof: constructive – we'll determine the corresponding cut C in G.

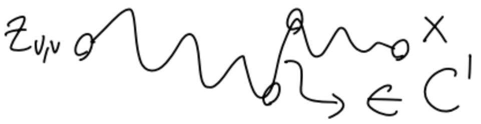
C' in $G/(e = (u, v))$ $\rightarrow C$ in G by substituting each edge $(z_{u,v}, y)$ in C' with the corresponding (u, y) or (v, y).





$|C'| = |C|$. It remains to be shown that C is a cut in G.

The hypothesis is the following one: C' is a cut in $G/e = (V', \mathcal{E}') \Rightarrow C'$ separates V' in 2 connected components; let $V_1 \subset V'$ the connected component containing $z_{u,v}$ and let $x \notin V_1$. Then, in G/e every path from $z_{u,v}$ and x must use an edge in C' (so, one which is not inside of V_1).



Now we'll show that C in G disconnects u and v from x: assume, by contradiction, that C is <u>not</u> a cut in $G \Rightarrow \exists$ a path between u and x after the removal of C in G (so, from $\mathcal{E}$). Then, the path between $z_{u,v}$ and x "survives" the removal of C' in G/e, i.e., C' is not a cut in G/e (because, if it survives, it means it does not use edges in C): *contradiction*.

- We have shown the contraction decreases the number of cuts
    a. (the cuts that disappear in the contracted graph are all and only those affected by e-side selection)
- That is, it can make some cuts disappear, certainly those affected by selection of e
- All others are preserved

This means that the only time the algorithm fails is when an edge belonging to a minimum cut in G is hit by the random choice.

- Basically, by proving the property above we have shown that
    a. If the algorithm never hits an edge belonging to a minimum cut
    b. Then it returns a correct solution (because the minimum cut is preserved in G/e)

What are the cuts that *disappear* in G/e? Those *hit by the random choice* ⇒ I want the probability of <u>not</u> hitting edges of the minimum cut to be *sufficiently high*.

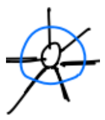
Intuition: $|min - cut|$ is a small fraction of $|\mathcal{E}|$. What $FULL\_CONTRACTION$ does is taking random edges hoping to not take a minimum-cut edge. We hope $|min - cut|$ is a small percentage of all the edges. This can be proven true easily.

Property: let a multigraph $G = (V, \mathcal{E}), |V| = n$. If G has a min-cut of size t, then $|\mathcal{E}| \geq t * \frac{n}{2}$

(Basically, the key property behind Karger's idea = number of edges is n times t compared to the cardinality of the minimum cut, so it can be much bigger)

Proof:

Consider the following multigraph, so here the notion of cut is visible:



By definition of minimum cut we have $d(v) \geq t$ for any vertex $v \in V$, where $d(v)$ denotes the degree of v. This is because if we take a node and remove all the edges incident to that node, that's a cut.

The sum of degrees is then split left and right, like follows:

$$\begin{tikzcd} d(v) \geq t \quad \forall v \in V \arrow[d, "\sum_{v\in V}"', "\sum_{v\in V}"] \\ 2|\mathcal{E}| \geq t * n \end{tikzcd}$$

Specifically:

$$\sum_{v \in V} d(v) = 2m \geq t * n$$

$$m = |E| \geq t * \frac{n}{2}$$

The analysis itself is based on conditional probabilities.

Definition: E_1, E_2 events are independent if $\Pr(E_1 \cap E_2) = \Pr(E_1) * \Pr(E_2)$

Definition: Given $\Pr(E_1) > 0$ then $\Pr(E_2|E_1) = \frac{\Pr(E_1 \cap E_2)}{\Pr(E_1)}$. This can be defined as the conditional probability, which refers to the chances that some outcome A occurs given that another event B has also occurred.

This can also be extended up to k events:

$$\Pr(E_1 \cap E_2 \cap \ldots \cap E_k) = \Pr(E_1)\Pr(E_2|E_1)\Pr(E_3|E_1 \cap E_2)\ldots\Pr(E_k|E_1 \cap \ldots \cap E_{k-1})$$

Such can be proven by induction on k. This will be used to evaluate the probability that $FULL\_CONTRACTION$ returns a minimum cut, where $E_i = i^{th}$ contraction which did not hit an edge of the min-cut.

Theorem: The probability that $FULL\_CONTRACTION$ returns a minimum cut in G is at least $\frac{2}{n*(n-1)}$

Proof: Although there may be > 1 min-cuts, we will actually prove that, for any min-cut C, the probability that the algorithm returns that particular min-cut C is at least $\frac{2}{n*(n-1)}$. So, let C be some specific min-cut.

Let's denote the size as $t = |C|$ and as E_i the variable meaning "in the i^{th} contraction I did not hit an edge in C" or better "avoid C"

- Observe that if only one minimum cut is considered, I am lowering the probability of success (I am okay with seeing the worst case).

We consider the event of a single shoot of the graph as the event E_1. Consider the complement event of the previous one: here, we will take a number of edges dependent on the actual vertices linked to that. More specifically:

$$\Pr(\overline{E_1}) = \frac{t}{|\mathcal{E}| \geq t * \frac{n}{2}} \leq \frac{t}{t * \frac{n}{2}} = \frac{2}{n}$$

With high probability, after the first "fire" session, the cut was not properly destroyed. We have to analyze what happens next, so the effect of the other iterations. We then use conditional probability in order to do so, which follows next here.

$$\Pr(E_1) = 1 - \Pr(\overline{E_1}) \geq 1 - \frac{t}{t * \frac{n}{2}} = 1 - \frac{2}{n}$$

E_2 is the probability that the second contraction avoid the minimum cut. This event is conditioned by the success of the previous; if C survived, it is inside a "transformed" cut (so, with different naming of the edges) of size $|C| = |C'| = t$.

We lost one vertex here because of contraction, so considering the previous, we have $(n-1)$ vertices here:

$$\Pr(E_2|E_1) \geq 1 - \frac{t}{t * \frac{(n-1)}{2}} = 1 - \frac{2}{n-1}$$

The probability starts to get a little lower this way so, going this direction, the probability of success will become lower and lower (given the graph gets smaller and there are only a few vertices).

$$\vdots$$

$$\Pr(E_i|E_1 \cap E_2 \cap \dots E_{i-1}) \geq 1 - \frac{t}{t * \frac{(n-i+1)}{2}} = 1 - \frac{2}{n-i+1}$$

$$\Pr\,(FULL\_CONTRACTION\ succeeds) = \Pr\,(\cap_{i=1}^{n-2} E_i) \geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right)$$

i.e., no edge of C is ever contracted

$$= \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1}$$

$$= \frac{\cancel{n-2}}{n} * \frac{\cancel{n-3}}{n-1} * \frac{\cancel{n-4}}{\cancel{n-2}} \dots \frac{\cancel{3}}{\cancel{5}} * \frac{2}{\cancel{4}} * \frac{1}{\cancel{3}} = \frac{2}{n(n-1)}$$

$$\geq \frac{2}{n^2}$$

We have shown, executing $FULL\_CONTRACTION$ only once, the probability of getting an edge not in the minimum cut is low, but not too low – given cuts in a graph may be even exponential in number!

So, eventually, this will return the min-cut. $KARGER$ amplifies this probability by repeating $FULL\_CONTRACTION$ k times (hence, correctly increasing and amplifying the probability).

We have to compute the probability the k iterations do not accumulate the size of the min-cut – so, none of the k runs returns the minimum cut.

$\Pr\,(the\ k\ runs\ of\ FULL\_CONTRACTION\ do\ not\ return\ the\ min\ cut) \leq \left(1 - \frac{2}{n^2}\right)^k \leq \frac{1}{n^d}$ for some constant $d > 0$

The previous one is the probability of an unsuccessful event, so we want it very low, something like $\frac{1}{n^d}$.

We want to find a value for k such that $\left(1-\frac{2}{n^2}\right)^k \leq \frac{1}{n^d}$. In this case, it's standard the use of this inequality:

$$\left(1+\frac{x}{y}\right)^y \leq e^x, y \geq 1, y \geq x$$

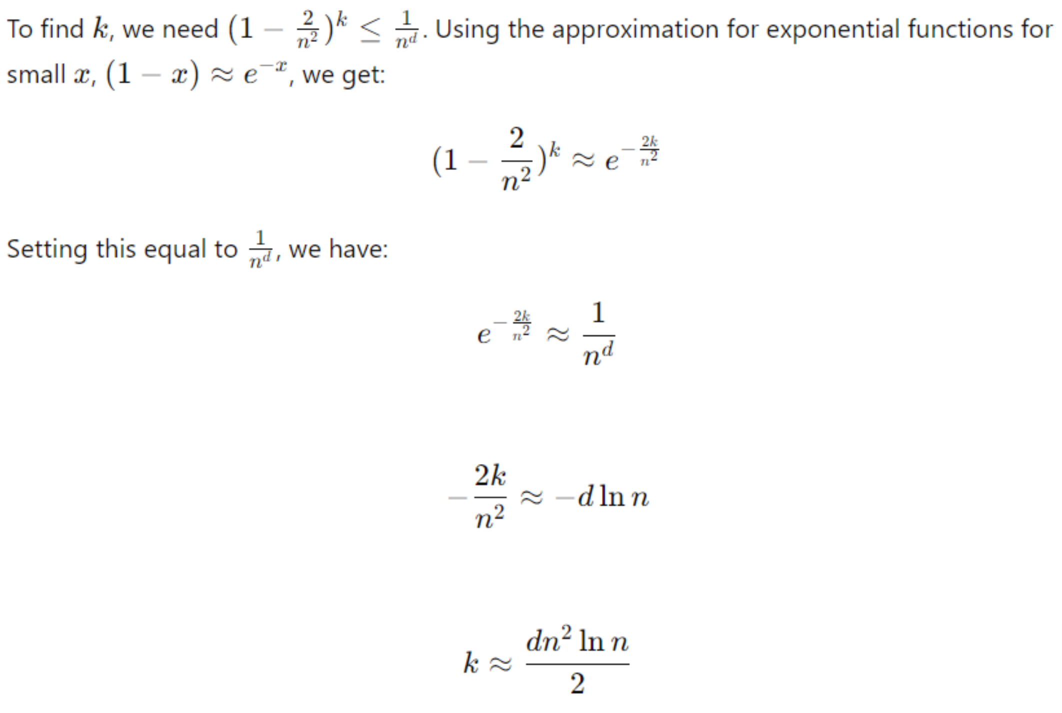
This inequality is derived from the exponential function and the binomial expansion. It represents an upper bound on the expression $\left(1+\frac{x}{y}\right)^y$, showing that it grows slower than e^x. The probability of not contracting the minimum cut in each iteration needs to be bounded and manipulated to ensure the overall algorithm's success probability is high.

By choosing $k = \frac{dn^2 \ln(n)}{2}$ it follows that:

$$\left(\left(1-\frac{2}{n^2}\right)^{n^2}\right)^{\ln(n^d)} \leq e^{-\ln(n^d)} = \frac{1}{n^d}$$

Given I am curious, I asked myself: why exactly that value for k?

Consider the probability of success if $\frac{2}{n^2}$ while the failure is, by complement, $1-\frac{2}{n^2}$ which, amplified by k runs, becomes $\left(1-\frac{2}{n^2}\right)^k$. The constant d is the desired level of confidence to keep the wanted threshold (in this case $\frac{1}{n^d}$) as low as possible. Then, using some good old GPT-4:



To find k, we need $(1-\frac{2}{n^2})^k \leq \frac{1}{n^d}$. Using the approximation for exponential functions for small x, $(1-x) \approx e^{-x}$, we get:

$$(1-\frac{2}{n^2})^k \approx e^{-\frac{2k}{n^2}}$$

Setting this equal to $\frac{1}{n^d}$, we have:

$$e^{-\frac{2k}{n^2}} \approx \frac{1}{n^d}$$

$$-\frac{2k}{n^2} \approx -d \ln n$$

$$k \approx \frac{dn^2 \ln n}{2}$$

Moving on:

$$\left(1-\frac{2}{n^2}\right)^{k=n^2} \leq e^{-2} = \frac{1}{e^2} \rightarrow \text{is } \underline{\text{not}} \text{ in the form } \frac{1}{n^d}$$

Recall the following:

$$e^{-\ln(n^d)} = \frac{1}{n^d}$$

Let's apply that:

$$\left(\left(1-\frac{2}{n^2}\right)^{n^2}\right)^{\ln(n^d)} = \left(1-\frac{2}{n^2}\right)^{n^2 \ln(n^d)}$$

Let's wrap up (here, in the prof. notes, d magically disappears, but I assume it to be 1 so this works):

$$\left(1-\frac{2}{n^2}\right)^{\boxed{k=\frac{dn^2\ln(n^d)}{2}}} = \left(\left(1-\frac{2}{n^2}\right)^{n^2}\right)^{\frac{\ln(n^d)}{2}}$$

$$\leq \left(e^{-2}\right)^{\frac{\ln(n^d)}{2}} = e^{-\ln(n^d)} = n^d = \frac{1}{n^d}$$

Then, by choosing that value for k the Karger's algorithm succeeds with high probability:

$$\Rightarrow \Pr(KARGER\ succeeds) > 1 - \frac{1}{n^d}$$

So, in the end, the Karger algorithm accumulates the size of the min-cut with probability at least $\frac{1}{n^d}$.

Complexity: $FULL\_CONTRACTION$ by itself has a complexity of $O(n * n) = O(n^2)$. Karger's algorithm has complexity $O(n^4 \log(n))$.

This can be improved with speeding up Karger's algorithm using the *Karger-Stein* variant (1996), using a stack to shrink recursively.

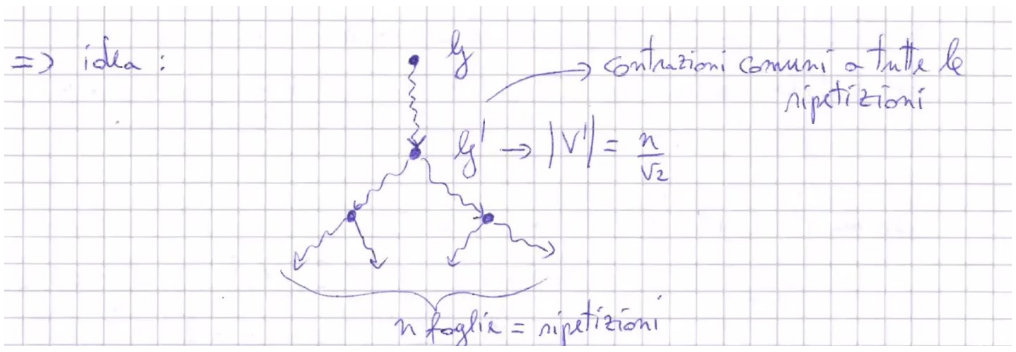
The idea is the following: $\Pr\,(failure\ of\ FULL\_CONTRACTION) = \underset{\underset{\text{1st contraction}}{\uparrow}}{\frac{2}{n}} \rightarrow \underset{\underset{\text{2nd}}{\uparrow}}{\frac{2}{n-1}} \rightarrow \underset{\underset{\text{3rd}}{\uparrow}}{\frac{2}{n-2}}$

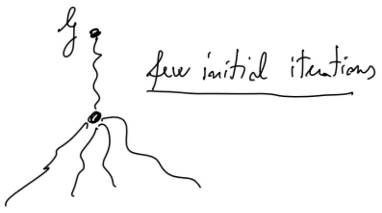
do not repeat the first $\sim \frac{n}{\sqrt{2}}$ contractions

In the first few iterations I have a very good probability of error, while as it goes on, the probability increases.

- The idea is to keep as contractions, common to all $FULL\_CONTRACTION$ repetitions, the first iterations of $FULL\_CONTRACTION$ because there is no point in repeating them
    a. These are already very good iterations
    b. and I will consider them to be the initial iterations for all the repeated $FULL\_CONTRACTIONS$ that I will do later

So, having common contractions for all repetitions, we will save a lot of work, like years ago would characterize:





This way, we will not repeat the first $\sim \frac{n}{\sqrt{2}}$ iterations → $O(n^2 \log^3(n))$ and correct w.h.p.

The current fastest algorithm has complexity $O(m(\log(n))$ (world record set in 2020).

So, in summary:

- Karger's algorithm effectively finds the minimum cut of a graph by repeatedly contracting edges and running multiple iterations to increase the probability of success
- By using conditional probability analysis, we can show that the algorithm succeeds with high probability when run sufficiently many times
- The key to the algorithm's efficiency lies in its use of randomness and the ability to amplify success probability through repeated trials

11.4.1 Exercises

Programming exercise

Implement Karger's algorithm and compare it with a deterministic algorithm.

Solution

```
import random

from copy import deepcopy

def karger_min_cut(graph):

""" Implements Karger's algorithm to find the minimum cut in an
undirected graph.

Args: graph (dict): A dictionary representing the graph, where keys are
vertices and values are lists of adjacent vertices.

Returns: int: The size of the minimum cut found. """ def contract(graph,
u, v): """ Contracts the edge (u, v) in the graph.

"""

new_graph = deepcopy(graph)

for neighbor in new_graph[u]:

if neighbor != u and neighbor != v:
```

```
new_graph[v].append(neighbor)

new_graph[neighbor].remove(u)

new_graph[neighbor].append(v)

del new_graph[u]

 new_graph[v] = [w for w in new_graph[v] if w != v]

return new_graph

# Repeat Karger's algorithm n^2 * log(n) times to increase the
probability of finding the minimum cut

n = len(graph)

min_cut_size = float('inf')

for _ in range(n**2 * int(max(1, round(2 * n * (n + 1) / 2)))):

curr_graph = deepcopy(graph)

while len(curr_graph) > 2:

u = random.choice(list(curr_graph.keys()))

v = random.choice(curr_graph[u])

curr_graph = contract(curr_graph, u, v)

 min_cut_size = min(min_cut_size,
len(curr_graph[next(iter(curr_graph))]))

return min_cut_size
```

This implementation repeatedly contracts edges in the graph until only two vertices remain. The size of the cut represented by the edges between these two vertices is a candidate for the minimum cut. The algorithm is repeated n^2 * log(n) times to increase the probability of finding the minimum cut, as suggested by the analysis of the algorithm.

To compare Karger's algorithm with a deterministic algorithm, we can implement a simple deterministic algorithm that enumerates all possible cuts and finds the minimum. Here's an implementation of such an algorithm:

```
def deterministic_min_cut(graph):

    """

    Finds the minimum cut in an undirected graph by enumerating all
possible cuts.

    Args:

        graph (dict): A dictionary representing the graph, where keys are
vertices

        and values are lists of adjacent vertices.

    Returns:
        int: The size of the minimum cut found.
    """
    def enumerate_cuts(graph, visited, curr_cut):
        if len(visited) == len(graph):
            cuts.append(curr_cut)
            return
        for vertex in graph:
            if vertex not in visited:
                new_visited = visited.copy()
                new_visited.add(vertex)
                new_cut = curr_cut.copy()
                for neighbor in graph[vertex]:
                    if neighbor not in new_visited:
                        new_cut.add((vertex, neighbor))
                enumerate_cuts(graph, new_visited, new_cut)
    cuts = []
    enumerate_cuts(graph, set(), set())
    min_cut_size = min(len(cut) for cut in cuts)
    return min_cut_size
```

This algorithm uses a recursive function to generate all possible cuts in the graph. It maintains a set of visited vertices and a set of edges that form the current cut. At each step, it considers adding an unvisited vertex to the visited set and updates the current cut accordingly. Once all vertices have been visited, the current cut is added to the list of cuts. Finally, the algorithm finds the minimum cut size from the list of all cuts.

While the deterministic algorithm guarantees to find the minimum cut, its time complexity is exponential in the number of vertices, making it impractical for large graphs. Karger's algorithm, on the other hand, has a running time of $O(n^2 * m)$, where n is the number of vertices and m is the number of edges. Although Karger's algorithm does not guarantee to find the minimum cut, it has a high probability of finding the minimum cut when repeated sufficiently many times.

In practice, Karger's algorithm is preferred for large graphs due to its efficient running time and probabilistic guarantee of finding the minimum cut. The deterministic algorithm can be useful for small graphs or as a reference implementation to verify the correctness of other algorithms.

<u>Exercise</u>

Using the analysis of Karger's algorithm, show that the n° of distinct min-cuts in a graph is at most $\frac{n(n-1)}{2}$*. Also, show that this bound is tight.*

Written by Gabriel R.

Solution

Let $C_1, C_2, \dots, C_j$ denote the min-cuts of a graph.

We have shown that $FULL\_CONTRACTION$ returns a particular min-cut C_i with probability $\geq \frac{2}{n(n-1)}$.

So, if we denote with A_i the event that C_i is returned by $FULL\_CONTRACTION$, we can write:

$$\Pr(A_i) \geq \frac{2}{n(n-1)}$$

Given the probability of the union of this event cannot be greater than 1, this term will not be that high. Observe that events $A_1, A_2, \dots A_j$ are disjoint. Then:

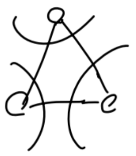
$$\Pr(A_1 \cup A_2 \cup \dots \cup A_j) = \sum_{i=1}^{j} \Pr(A_i)$$

By definition, $\Pr(A_1 \cup A_2 \cup \dots \cup A_j) \leq 1$, so $\sum_{i=1}^{j} \underbrace{\Pr(A_i)}_{\geq \frac{2}{n(n-1)}} \leq 1 \Rightarrow j \leq \frac{n(n-1)}{2}$

This bound is tight: in a cycle of n vertices every pair of edges is a distinct min-cut:

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

$n = 3$



$$\leq \frac{3 \cdot \cancel{2}}{\cancel{2}} = 3$$

12 CHERNOFF BOUNDS

Chernoff bounds are tools from modern probability theory that are frequently used in the analysis of randomized algorithms. They're a more powerful version of the Markov's lemma.

Consider the phenomenon of "concentration of measure":

- *Toss a coin:*
  a. 1 time → outcome is unpredictable
  b. 1000 times → outcome is sharply predictable!
- *Application*
  a. $T(n)$ guaranteed to be concentrated around some value

In many cases, the study of $\Pr\,(T(n) > c * f(n))$ can be rephrased of the study of the distribution of a sum of random variables. In this course, we will talk about indicator random variables (also called $0-1$ random variables or even Poisson trials), which map every outcome to either 0 or 1. This can be represented as:

$$r = \begin{cases} 1, & if\ trial\ is\ a\ success \\ 0, & otherwise \end{cases}$$

In general, this can be represented as:

$$X = \sum_{i=1}^{n} X_i \qquad X_i\ indicator\ random\ variables$$

The variable X counts the number of successes (generally, the outcome) obtained by the n indicator variables and X_i is the indicator random variable associated to a specific event. We'll usually have that X_i's are independent between each other ($=^{\Delta}$ means "equal by definition").

$$\Pr(X_1 = 1) = p_i$$

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} p_i =^{\Delta} \mu$$

(the previous sum is the average of the sum of the variables X_i)

We analyze the probability that X deviates from $E[X]$ (so, from the average and this is very low):

$$\Pr(X > (1+\delta)\mu) \leq \frac{E[X]}{(1+\delta)\mu} = \frac{\mu}{(1+\delta)\mu} = \frac{1}{1+\delta}$$

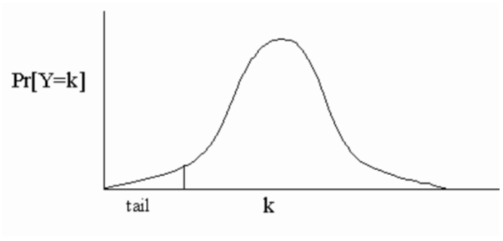
Markov → $\Pr(X \geq t) \leq \frac{E[X]}{t}$

Usually, using Markov inequality, this is not a very good bound.

That is, the probability of a certain event X deviating from its mean is very low. To be perfectly precise:

- It is a sharper bound than the first- or second-moment-based tail bounds such as Markov's inequality or Chebyshev's inequality, which only yield power-law bounds on tail decay
- However, when applied to sums the Chernoff bound requires the random variables to be independent, a condition not required by either Markov's inequality or Chebyshev's inequality

Generally, Chernoff bounds are a tool which allow to study the concentration of an event around its mean (specifically, in the "tail" – see figure – so, far from the mean) and to overcome the previous fact we use them. Specifically, they provide exponentially decreasing bounds on the tail distributions of sums of independent random variables.



- The markup is a little loose, not very significant
- Better augmentation allows me to move from analysis to the average case to the more desirable high probability analysis

The idea between Chernoff bounds is to transform the original random variable into a new one, such that the distance between the mean and the bound we will get is significantly stretched. It answers the question about how tight the bound we can get when having more information about the distribution of the random variables.

12.1 Overview: A more powerful probabilistic tool

We give *Chernoff's lemma* here (Chernoff bound): let $X_1, X_2, \dots X_n$ independent indicator random variables where $E[X_i] = p_i, 0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then $\forall\ \delta > 0$:

$$\Pr(X > (1+\delta)\mu) < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

This is known as "Multiplicative Chernoff Bound".

In words: the outcome concentrates around the min is very high – to the contrary, the probability of deviating from the min should be very low. For details about the proof see here (third page) or see "Probability and Computing" book in the related "Chernoff Bounds" chapter.

Consider the example of coin tossing: there are n coin flips, so characterized by $X_1, X_2, \dots X_n$ (of course, the coin is fair):

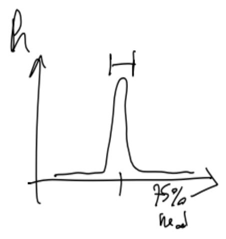
$$\Pr(\underset{\overset{\parallel}{\text{head}}}{X_i = 1}) = \frac{1}{2} \quad \forall i$$

Basically, $X_1 = 1$ means "get *heads*", otherwise $X_i = 0$ means "get *tails*"

$$X = \sum_{i=1}^{n} X_i = n^{\circ}\ of\ heads\ in\ n\ coin\ flips$$

Specifically, the sum counts the number of heads (*successes*) obtained by flipping n times the coin – so, I expect to get $\frac{n}{2}$ heads.

$$E[X] = \frac{n}{2}$$



Question: What's the probability of getting more than $\frac{3}{4}n$ heads? Veeery low.

Here, both Markov and Chernoff lemmas will be applied.

1) Markov's lemma

$$\Pr\left(X > \frac{3}{4}n\right) \le \frac{E[X]}{\frac{3}{4}n} = \frac{\frac{n}{2}}{\frac{3}{4}n} = \underbrace{\frac{2}{3}}_{\text{constant}}$$

2) Chernoff bound

$$\underbrace{\Pr\left(X > \left(1 + \frac{1}{2}\right)\mu\right)}_{= \frac{3}{4}n} < \left(\frac{e^{\frac{1}{2}}}{\left(\frac{3}{2}\right)^{\frac{3}{2}}}\right)^{\frac{n}{2}} < \underbrace{(0.95)^n}_{\text{exponential!}}$$

All of this is saying, instead of using only Markov, here we will have a much tighter bound (for $n \to \infty$ the probability goes to 0 or approximates to 0 really fast), that's why it's this powerful. Here, we passed from a constant probability to an exponential probability.

12.2 Chernoff Bound Variants

Consider the following *variants of Chernoff bounds* (weaker but easier to state and to use):

1) $\Pr(X < (1-\delta)\mu) < e^{-\frac{\mu\delta^2}{2}}, 0 < \delta \le 1$ (Lower tail bound)

2) $\Pr(X > (1+\delta)\mu) < e^{-\frac{\mu\delta^2}{2}}, 0 < \delta \le 2e - 1$ (Upper tail bound)

Quoting the professor: you do not have learn by heart, because during an exam they are given by the professor, we do not need to learn them. So, also quoting older Italian notes: in the exam all 3 variants (the normal one and these ones) + Markov's lemma will be given. Just check the MEGA in case.

12.3 Analysis in High Probability of Randomized Quicksort

(Further readings for this one: here)

As an example of application of Chernoff bounds, we do the analysis of Randomized Quicksort, in which we remember the pivot is chosen at random (possibly, not very far from the median) and partitions the array around the pivot. This random selection of pivots ensures that the algorithm performs well on average, even though the worst-case complexity can be poor. This is a LAS VEGAS algorithm, since it always sorts.

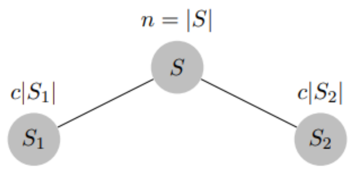
RQS:
1) pick a "pivot" element at random
2) split the array into 3 subarrays <
   - smaller than pivot
   - pivot itself
   - larger than pivot
3) recursively sort the subarrays and then concatenate them

```
procedure $RandQuickSort(S)$                    $|S| = n$, all distinct
    if $|S| \leq 1$ then return $\langle S \rangle$
    $p = RANDOM(S)$    $// pick\ a\ \text{“}pivot\text{”}\ element\ uniformly\ at\ random\ from\ S$
    $S_1 = \{x \in S\ s.t.\ x < p\}$    $// O(n)\ time$
    $S_2 = \{x \in S\ s.t.\ x > p\}$    $// O(n)\ time$
    $Z_1 = RandQuickSort(S_1)$
    $Z_2 = RandQuickSort(S_2)$
    return $\langle Z_1, p, Z_2 \rangle$
```

The complexity of $RandQuickSort$ depends on how well the pivot divides the array into two parts. If the pivot is close to the median, the partitions will be well-balanced, leading to efficient sorting.

Suppose p is always the median of S; then

$$T_{RQS}(n) = \begin{cases} 2T_{RQS}\left(\frac{n}{2}\right) + O(n), & n > 1 \\ 0, & n \leq 1 \end{cases}$$



So, basically:

- There are n nodes (excluding leaves associated to $\emptyset$) $\Rightarrow \leq n$ paths root-leaf
- We will show these paths are not so long; using any method (Master theorem or whatever), we get the same height of the tree

$$T_{RQS}(n) =^{Master\ Theorem} O(n\log(n))$$

However, p is the median with probability $\frac{1}{n}$, which is very low.

(Can we find the median in linear time? Yes, deterministically $\Rightarrow$ deterministic QS can be done in $O(nlog(n))$, but the algorithm is very complicated, and the hidden constant (from Big O notation) is very high, thus inefficient in practice).

But do we really want exactly the median?

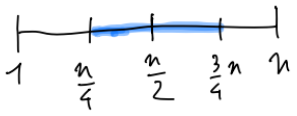
Intuition: if the sizes of S_1 and S_2 are “not too unbalanced”, we should be good. Fortunately, we don’t really need that the random choice hits always the median. We assume the pivot splits the array such that the sizes of the partitions are not too unbalanced.

Let's try anyway with a loose request for the approximation:

$$\begin{cases} |S_1| \leq \frac{3}{4}n \\ |S_2| \leq \frac{3}{4}n \end{cases}$$

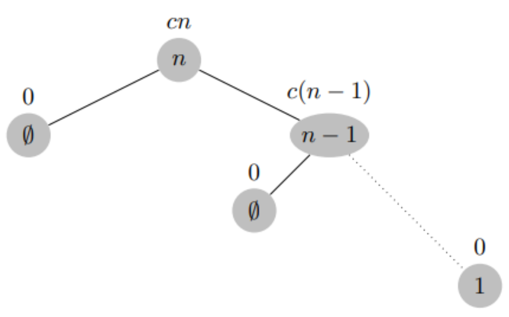
Conversely, note that it holds also, $|S_1| \geq \frac{n}{4}, |S_2| \geq \frac{n}{4}$.

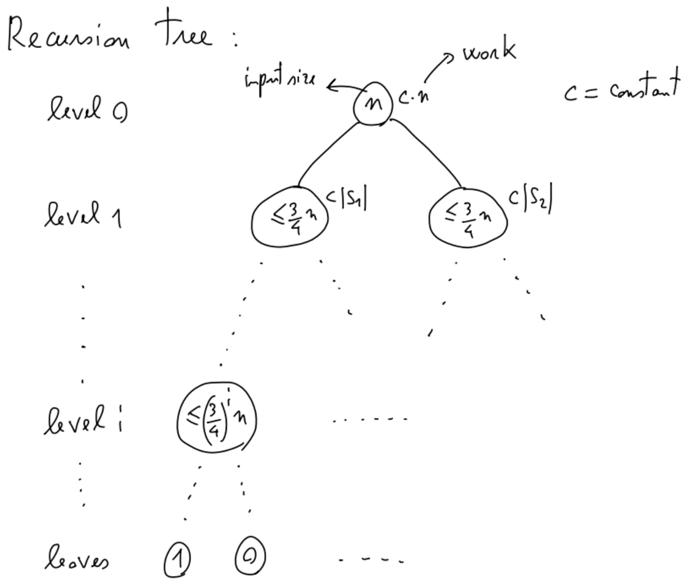
That is, the pivot p chosen from:



Can this be considered good enough for us?

Let's assume we always choose the pivot in between this range. This can be easily seen via a *recursion tree*. We want to show that the depth of the recursion tree is $O(\log(n))$ with high probability. This means that the height of the tree, representing the number of recursive calls from root to leaf, is logarithmic in the size of the input in the worst case:





What is going on here? Every choice of the algorithm splits the size in at most $\frac{3}{4}n$, given the chance of an unlucky choice, given the subtrees can be unbalanced and the subinstances can be at most that. The generic event E can be characterized as the "good choice" of the pivot between all the statistically possible choices. So, he good pivot is chosen such that the size of the partitions are at most $\frac{3}{4}n$.

So, calculating the cost of the algorithm:

- Total work at each level is mostly linear, so $\leq c * n$
- Depth of the recursion tree $= \min\left\{integers\ i\ s.t.\left(\frac{3}{4}\right)^i n \leq 1\right\} = \lceil \log_{\frac{4}{3}}(n) \rceil = O(\log(n))$

  So, continuing: $\left(\frac{3}{4}\right)^i n \leq 1 \Leftrightarrow \left(\frac{3}{4}\right)^i \leq \frac{1}{n} \Leftrightarrow \left(\frac{4}{3}\right)^i \geq n \Leftrightarrow \log_{\frac{4}{3}}\left(\frac{4}{3}\right)^i \geq \log_{\frac{4}{3}}(n) \Leftrightarrow i \geq \log_{\frac{4}{3}}(n)$

$\Rightarrow T_{RQS}(n) = O(n\, log(n))$

By looking at the recursion tree, if such a pivot is always chosen, all the root-leaf paths are no longer that the factor obtained (specifically, $\log_{\frac{4}{3}}(n)$), given after i success, in S we have $\leq \left(\frac{3}{4}\right)^i n$.

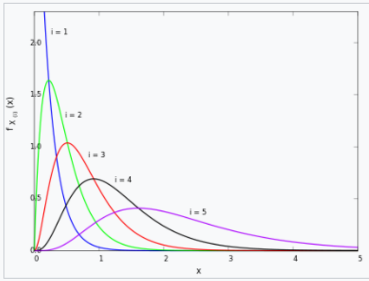
That is, it's not necessary that S_1 and S_2 are perfectly balanced. Specifically, we have $\simeq \frac{n}{2}$ "good" choices for the pivot p.

12.3.1 Analysis

Hope: depth of the recursion tree $= O(\log(n))$ w.h.p. That is, all the $\leq n$ distinct root-leaf paths have $O(\log(n))$ length w.h.p. Let's call the event E = "lucky choice of the pivot", that is, pivot chosen between the $(\frac{n}{4}+1)$-th order statistic and the $(\frac{3}{4}n)$-th order statistic – so, a range around the median point, possibly large. This means the 2 subinstances have size $\leq \frac{3}{4}n$.

(Unrelated, but good to see the point here)

Quoting Wikipedia, we have that In statistics, the k^{th} order statistic of a statistical sample is equal to its k^{th} -smallest value.



Probability density functions of the order statistics for a sample of size n = 5 from an exponential distribution with unit scale parameter

$$\Pr("lucky\ choice") = \frac{\frac{3}{4}n - \left(\frac{n}{4}+1\right)+1}{n} = \frac{1}{2}$$

If I always have success, the paths are no longer than $\log_{\frac{4}{3}} n$ (because $|S|$ after the successes is $\leq \left(\frac{3}{4}\right)^i n$). Fix one root-leaf path P and the following lemma says, "with h.p. the path chosen in short"; specifically shorter than $\log(n)$".

Lemma: $\Pr(|P| > a * \log_{\frac{4}{3}}(n)) < \frac{1}{n^3}$

$\hookrightarrow$ constant

We are trying to find the probability there is at least one big path around the mid value.

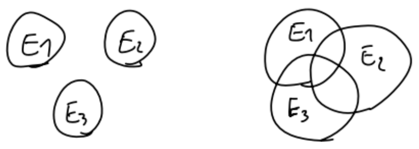
If this is true, we're done, applying the very frequent/very famous following lemma.

Lemma (Union bound): for any random events $E_1, \dots E_K$:

$$\Pr(E_1 \cup E_2 \cup \dots \cup E_k) \leq \Pr(E_1) + \Pr(E_2) + \cdots + \Pr(E_k)$$

(the probability of one of the events happening, given they are disjointed, is no more than the original sum). Consider the picture: given all independent events, the probability of the union is no more than the union of all cases.

proof by picture :



In the context of analyzing randomized algorithms with Chernoff bounds, the union bound helps us manage the probability that any one of multiple bad events occurs. This way, we can ensure that our randomized algorithms perform well not just in isolated instances, but across the entire set of possible outcomes.

So, we want to prove the following:

$$\Pr\left(all\ root - leaf\ paths\ have\ length \leq a * \log_{\frac{4}{3}}(n)\right) \geq 1 - \Pr\left(\exists\ path > a * \log_{\frac{4}{3}}(n)\right) \geq \cdots$$

If the lemma is true, it follows that:

- Given the event E_i = the path p_i has length $> a * \log_{\frac{4}{3}}(n)$:

$$\Pr\left(\exists\ path > a * \log_{\frac{4}{3}}(n)\right) = \Pr\left(\bigcup_{i=1}^{n} E_i\right) \leq^{union\ bound}$$

$$\leq \sum_{i=1}^{n} \Pr(E_i) <_{lemma} n * \frac{1}{n^3} = \frac{1}{n^2}$$

$$\ldots \geq 1 - \frac{1}{n^2}$$

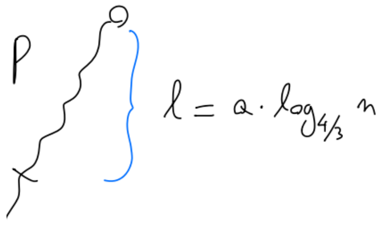
$\Rightarrow T_{RQS}(n) = O(n\log(n))$ w.h.p. (basically, all paths are short than the factor, so $1 - \frac{1}{n^2}$)

It remains to prove the lemma, so to prove that:

$$\Pr\left(|P| > a * \log_{\frac{4}{3}}(n)\right) < \frac{1}{n^3}$$

↳ constant

Specifically, in words, a fixed path has more than l, as follows:



So in the course of pivot choices, about one out of two times a lucky choice is made. The intuition is that, on average, a path will be no more than twice as long as the shortest possible path. Along a root-to-leaf path, every second node guarantees a $\frac{3}{4}$ reduction in the size of the set.

This guarantees a logarithmic number of levels, corresponding to an $n(\log(n))$ algorithm with high probability, a with a pivot choice that can be realized in constant time.

The event E can be characterized as "in the first $l = a * \log_{\frac{4}{3}}(n)$ nodes of P there have been $< \log_{\frac{4}{3}}(n)$ lucky choices". We are studying this specific event:

- $X_i, 1 \le i \le l = a * \log_{\frac{4}{3}}(n)$
- $X_i = 1$ if at the i^{th} vertex of P there is a lucky choice of the pivot
- $\Pr(X_i = 1) = \frac{1}{2}\ \forall i$
- X_i are independent

We want the probability of $P\left(\sum_{i=1}^{l} X_i < \log_{\frac{4}{3}}(n)\right)$ to bound $X = \sum_{i=1}^{l} X_i$. Given $X = \sum_{i=1}^{l} X_i$, its expected value is as follows:

$$\mu = E[X] = E[\sum_{i=1}^{l} X_i] = \sum_{i=1}^{l} E[X_i] = \sum_{i=1}^{l} \frac{1}{2} = \frac{l}{2} = \frac{a}{2}\log_{\frac{4}{3}}(n)$$

Now, let's apply the following Chernoff bound (the first):

$$\Pr(X < (1-\delta)\mu) < e^{\frac{-\mu\delta^2}{2}}, 0 < \delta \le 1$$

$$\downarrow$$

$$(1-\delta)\mu = \log_{\frac{4}{3}}(n)$$

$$(1-\delta)\frac{a}{2}\log_{\frac{4}{3}}(n) = \log_{\frac{4}{3}}(n)$$

One possible choice is $a = 8, \delta = \frac{3}{4}$, obtained using symbolic analysis and finding the right values to respect the conditions above:

$$t = 8\log_{4/3} n \quad \rightarrow \quad \mu = 4\log_{4/3} n$$
$$\delta = \frac{3}{4} \quad \rightarrow \quad 1 - \delta = \frac{1}{4}$$
$$(1-\delta)\,\mu = \log_{4/3} n$$

We then apply the Chernoff lemma as follows:

$$\Pr\,(X < \log_{\frac{4}{3}}(n)) < e^{-\frac{8}{4}*log_{\frac{4}{3}}(n)*\frac{9}{16}}$$

$$= e^{-\frac{8}{4}*log_{\frac{4}{3}}(n)*\frac{9}{8}}$$

$$< e^{-log_{\frac{4}{3}}(n)}$$

$$= e^{-\frac{ln(n)}{ln\left(\frac{4}{3}\right)}}$$

$$= \left(e^{-ln(n)}\right)^{\frac{1}{\ln\left(\frac{4}{3}\right)}}$$

$$= \left(\frac{1}{n}\right)^{\frac{1}{\ln\left(\frac{4}{3}\right)} \simeq 3{,}47}$$

$$< \left(\frac{1}{n}\right)^{3{,}47} = \left(\frac{1}{n^{3{,}47}}\right) \simeq \frac{1}{n^3}$$

So, we have:

$$\Pr\left(\sum_{i=1}^{l} X_i < \log_{4/3} n\right) < \frac{1}{n^3}$$

Using the union bound over all n paths:

$$\Pr(\text{all paths are short}) \geq 1 - n \cdot \frac{1}{n^3} = 1 - \frac{1}{n^2}$$

Or equivalently (left in Italian since it's mostly a detailed analysis):

Riassumendo, l'evento "cattivo" per cui l'albero ha altezza maggiore di $a \log_{4/3} n$ accade con probabilità

$$Pr\left(\text{l'albero ha altezza} > 8\log_{4/3} n\right) \leq$$

$$\begin{aligned}
&= Pr\left(\text{esiste } \textit{un} \text{ cammino di lunghezza} > 8\log_{4/3} n\right)\\
&\leq nPr\left(\text{un cammino } \textit{fissato} \text{ ha lunghezza} > 8\log_{4/3} n\right)\\
&\leq nPr\Big(\text{ci sono meno di } \log_{4/3} n \text{ scelte fortunate}\\
&\qquad\qquad \text{nei primi } a\log_{4/3} n \text{ nodi del cammino}\Big)\\
&\leq n\frac{1}{n^3} = \frac{1}{n^2}
\end{aligned}$$

Hence, in high probability, the randomized quicksort recursion tree has a logarithmic number of levels, and each level contributes to the overall work $O(n)$, so in high probability quicksort performs in $O(n(\log(n)))$ time.

For randomized algorithms, the analysis is decomposed into a series of events that may or may not happen, and then studies the good (or bad) event as a function of these individual elementary events that capture the various choices made by the algorithm.

Deterministic analysis is similar; you break down the algorithm into elementary steps and find what the worst-case sequence is. Rather than the worst case, here one studies the probability of that happening, decomposing the entire computation into a series of events to be composed of each other in a way that uses the known bounds.

(This year, the program ends here – next classes will be only exercises dedicated for the exam. Information for you to be organized upfront – since no classes were skipped, here this end of program is Half of May, so there's time to prepare yourself).

12.4 Applications of Chernoff Bounds

(Note: these two lessons the professor will say "program is over, and we will do lessons dedicated to exercises for the exam". Point is, this is not entirely true, since these are more of use cases which effectively appeared inside of exams (see here) but is just the same script year by year.

12.4.1 Exit polls

The first to be analyzed here is exit polls: approximate the percentage (%) of voters that in an election voted for one of the available options, without counting all votes. More generally, as noted here, this can be applied to sampling/polling cases.

How does it work?

- Some votes are drawn at random, which will go to represent the approximate solution
- The idea is that *enough* votes should be drawn
    - Which assures me the goodness of the solution w.h.p.

One urn U (container) with n balls (which represent parties), both white and black.

Goal: approximate the true value of white balls $\alpha * n$.

Assumption: we know that there are $\alpha_{min} * n$ white balls (we have to assume it for certain, because it is difficult to approximate w.h.p. a very improbable event). That is why, on small parties, we get the % that is given to exit polls much more wrong.

To determine α:

- With a *deterministic* algorithm (exact), the complexity is $\Omega(n)$
- With a *randomized approximated* algorithm, the complexity if $O(\log(n))$
    - And that's why we can do exit polls!

The algorithm will output a quantity (estimate) β such that $\Pr\left(\frac{|\beta-\alpha|}{\alpha} > \epsilon\right)$ is very low (and so to determine the number of samples k to ensure estimate β is close to α with high probability):

- $\frac{|\beta-\alpha|}{\alpha}$ is the *relative error* – measure of difference between approximate value and true value
- ϵ is the *confidence threshold* – predetermined probability level defining the minimum acceptable likelihood for the event to be considered significant or true
    - an example of very low value is e.g., $< \frac{1}{n^2}$.

Note: it's a randomized approximation scheme. The smaller ϵ is, the longer it will take the algorithm, because it increases the number of extractions that will be necessary to do.

```
$APPROXIMATE\_\alpha\ (U, \epsilon, \alpha_{min})$
    $n = |U|$          // balls present
    $k = f(n, \epsilon, \alpha_{min})$        // n° of extractions to be determined in the analysis
    $x = 0$
    repeat $k$ times
        $p = RANDOM(U)$
        if $color(p) = white$ then $x{+}{+}$
    return $\frac{x}{k}$
        $\downarrow$
        $\beta$ (value which approximates $\alpha$)
```

Note that these are extractions with reintroduction, whereas in exit polls the extractions are without reintroduction. This only improves the approximation of the algorithm because the extractions without reintroduction are always worse because they have greater variance.

So, to summarize:

- Use Chernoff bounds to determine the number of extractions k needed to guarantee that the estimate $\frac{x}{k}$ is close to p with high probability

Complexity is (very fast dependent on the number of samples) → $O(k)$.

Main question: "What's the value of k that guarantees the high probability"?

- k indicator random variables
- $X_i = 1$ if the i^{th} extracted ball is white (of course, 0 otherwise)
- $\Pr(X_i = 1) = \alpha$ (the parameter is not known, fraction voting basically)
- $X = \sum_{i=1}^{k} X_i$, which is the n° of extracted white balls (estimate)
- $\mu = E[X] = E[\sum_{i=1}^{k} X_i] = \sum_{i=1}^{k} E[X_i] = k\alpha$

Event: (written in the form of approximate Chernoff bound - represents the case where our estimate β (which is $\frac{x}{k}$) differs from the true value α by more than a relative error ϵ.

$$"\frac{|\beta - \alpha|}{\alpha} > \epsilon" = "\frac{\left|\frac{X}{k} - \alpha\right|}{\alpha} > \epsilon"$$

$$= "\underbrace{\frac{|x - \alpha k|}{\alpha k}} > \epsilon"$$

After isolating the expected value, we'll use this Chernoff bound:

$$\Pr(|X - \mu| > \epsilon\mu) < 2\underbrace{e^{\frac{-\mu\epsilon}{2}}}_{\text{goal: } \sim \frac{1}{n^2}}, 0 < \epsilon \le 1$$

Issue: α is unknown $\Rightarrow$ use α_{min} instead (so, it's a lower bound for α and so $\Rightarrow$ use $a_{min} \leq \alpha$).

$$2e^{\frac{-k\alpha\epsilon^2}{2}} \leq 2e^{\frac{-k\alpha_{min}\epsilon^2}{2}} \to k = \frac{2}{n^2}$$

(the previous has to become $< \frac{1}{n}$).

Then, algebraic operations and solving from k:

$$-\frac{k\alpha_{min}\epsilon^2}{2} = -\ln(n^2) \to e^{-\ln(n^2)} = \frac{1}{n^2} \Rightarrow k = \frac{2\ln(n^2)}{\alpha_{min}\epsilon^2} = \frac{4\ln(n)}{\alpha_{min}\epsilon^2} O\left(\frac{\log(n)}{\epsilon^2}\right)$$

$$2e^{\frac{-k\alpha\epsilon^2}{2}} \leq 2e^{\frac{-k\alpha_{min}\epsilon^2}{2}} = 2e^{-\ln(n^2)} = \frac{2}{n^2} < \frac{1}{n} \quad (for\ n > 2)$$

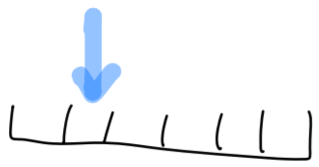
(if $k \uparrow$ I get $\frac{1}{n^d}$ with $d > 1$; here, k is chosen so to remove parameters different from n to add then a $\log(n)^\omega$ so to obtain something like $e^{-\log(n^\omega)}$, which is equal to $n^{-\omega} = \frac{1}{n^\omega}$)

So, we exponentially decreased the complexity of the deterministic algorithm maintaining a relative margin of error which w.h.p. is very small.

12.4.2 Load balancing

The problem is also called "balls-and-bins" and it can be described as follows:

- n servers
- n jobs/requests, that arrive one by one



There are some *issues* however:

- Distributed environment, so no central control
- Limited information, where we don't know the servers' loads (latency)

Goal: minimize max load over the n servers (side note: as before, this can be found on page 7 here).

Simple algorithm: assign each job to a server chosen uniformly at random.

- This is definitely more efficient than maintaining some state and/or statistics, which might cause slight delays – the policy is simple and lightweight

Does it work? The *general model* to be followed here is the "balls-and-bins" – classic CS problem of assigning m balls into n boxes/bins considering problem like this one or even hashing and fair cake-cutting (yeah, that's a thing) – so:

- Consider a fixed process (which in RQS meant considering a fixed root-leaf path) then make the analysis for that single element
- Apply a union bound (bound on the union probability of events) to obtain the probability whatever processor having a high load is very low

Theorem (famous result): if n inputs are assigned uniformly at random to n servers, then with probability $\geq 1 - \frac{1}{n}$ every server has $\leq \frac{3\ln(n)}{\ln(ln(n))}$ requests, assuming sufficiently high n.

Side note: Given I am not content with "famous result" now knowing where this comes from, it's from the balls and bins model but also a problem called "coupon's collector", which you can see here.

Proof:

- Consider a fixed server
  - $X_i = 1$ if the i^{th} job/request gets assigned to that server (one for every request)
  - $\Pr(X_i = 1) = \frac{1}{n}$
  - X_i's are independent
  - $X = \sum_{i=1}^{n} X_i$ = load of that server (quality to be analyzed)
  - $\mu = E[X] = \sum_{i=1}^{n} E[X_i] = n * \frac{1}{n} = 1$ = average number of requests of a server

Now, we will study X in high probability. We'll use the following (Chernoff lemma in original version, so the strongest version. If we apply the weaker ones, we obtain only a single ln factor, this does a tight approximation):

$$\Pr(X > (1+\delta)\mu) < \left(\frac{(e^{\delta})}{(1+\delta)^{1+\delta}}\right)^{\mu}$$

$$\downarrow$$

$$\frac{3\ln(n)}{\ln(\ln(n))} \Rightarrow \delta = \frac{3\ln(n)}{\ln(\ln(n))} - 1$$

What we want to claim exactly is that the probability of a server exceeding the specified load is at most $\frac{1}{n^2}$. Here, we apply the substitutions necessary for Chernoff bounds, using logarithmic/exponential properties:

$$\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \leq^{?} \frac{1}{n^2}$$

$$\Updownarrow \qquad \text{(take logs of both sides)}$$

$$\delta - (1+\delta)\ln(1+\delta) \leq -2\ln(n)$$

$$\Updownarrow$$

$$\frac{3\ln(n)}{\ln(\ln(n))} - 1 - \frac{3\ln(n)}{\ln(\ln(n))}\ln\left(\frac{3\ln(n)}{\ln(\ln(n))}\right) \leq -2\ln(n)$$

$$\Updownarrow$$

$$\frac{3\ln(n)}{\ln(\ln(n))} - 1 - \frac{3}{\ln(ln(n))}(\ln(3) + \ln(\ln(n)) - \ln(ln(ln(n))))) \leq -2\ln(n)$$

$$\Updownarrow$$

$$\frac{3\ln(n)}{\ln(\ln(n))} - \frac{1}{\ln(n)} - \frac{3}{\ln(ln(n))} - 3 + \frac{3\ln(\ln(\ln(n)))}{\ln(ln(n))} \leq -2$$

$$\Updownarrow \qquad n \text{ sufficiently high}$$

$$O(1) + O(1) + O(1) - 3 + O(1) \leq -2 ✓$$

(In words: the load is at most is definitely the quantity in high probability, so $< \frac{1}{n^2}$)

Now, let's apply the union bound (which allows to sum up for all the probabilities for every server and checks the probability of any server being overloaded is very low) to see that the same is true for every server simultaneously:

$E_i =$ the i^{th} server gets more than $\frac{3\ln(n)}{\ln(\ln(n))}$ requests

$$\Pr\left(\exists\ \textit{server that gets more that } \frac{3\ln(n)}{\ln(\ln(n))}\ \textit{requests}\right) = \Pr\left(\bigcup_{i=1}^{n} E_i\right) \leq_{union\ bound} \sum_{i=1}^{n} \Pr(E_i) = n * \frac{1}{n^2} = \frac{1}{n}$$

In other words, the probability that no server gets more than $\frac{3\ln(n)}{\ln(\ln(n))}$ jobs is $> 1 - \frac{1}{n}$.

(Consider in past notes, $\frac{3\ln(n)}{\ln(\ln(n))} = \theta\left(\frac{\log(n)}{\log(log(n))}\right)$)

Two notes in the end:

- It can be shown to be tight: some server gets $\Omega(\frac{\log(n)}{\log(log(n))})$ requests)
- Improved algorithm: choose 2 servers at random and assign the request to the least loaded → max load drops to $O(\log(\log(n))$!

The first one was present in Lecture 23 of 2022-2023, for the sake of completeness (and craziness, I'd say), I'll add it here anyway. This uses the original version of Chernoff lemma:

$$\Pr(X > (1+\delta)\mu) < \left(\frac{(e^{\delta})}{(1+\delta)^{1+\delta}}\right)^{\mu}$$

call $(1+\delta) = c$, then:

$$\Pr(X > c) < \frac{e^{c-1}}{c} < \left(\frac{e}{c}\right)^{c}$$

$$c = e\gamma(n) \rightarrow \left(\frac{e}{c}\right)^{c} = \left(\frac{1}{\gamma(n)}\right)^{e\gamma(n)} < \left(\frac{1}{\gamma(n)}\right)^{2\gamma(n)}$$

It holds that $\gamma(n)^{\gamma(n)} = n \Leftrightarrow \gamma(n) = \theta\left(\frac{\log(n)}{\log(\log(n))}\right)$, then:

$$\Pr\left(X > \theta\left(\frac{\log(n)}{\log(\log(n))}\right)\right) < \gamma(n)^{-2\gamma(n)} = \left(\gamma(n)^{\gamma(n)}\right)^{-2} = n^{-2} = \frac{1}{n^2}$$

Applying the union bound:

$E_i =$ the i^{th} server gets more than $\theta\left(\frac{\log(n)}{\log(\log(n))}\right)$ jobs

$$\Pr\left(\exists\ \textit{server that gets more than } \theta\left(\frac{\log(n)}{\log(\log(n))}\right)\ \textit{jobs}\right) = \Pr\left(\bigcup_{i=1}^{n} E_i\right) \leq_{Union\ Bound} \sum_{i=1}^{n} \Pr(E_i)$$

$$= n * \frac{1}{n^2} = \frac{1}{n}$$

In other words, the probability that no server gets more than $\frac{3\ln(n)}{\ln(\ln(n))}$ jobs is $> 1 - \frac{1}{n}$.

13 LAST LESSON - EXERCISES

(This lesson represents an example of exercises for the exam; I suggest to just reference to what is present here. Delving into past material, he proposed the same algorithm every year up to 22/23 but this year, so 23/24, he actually showed something different. For the sake of completeness, I showed here also the previous years example of exam exercise, present since 19/20, first year of this course)

This is the last class of the course. On the exam itself:

- Written test → 2 hours
- 2 parts
  - Theory questions (to verify that you know the program)
    - 3 theory questions (~ 4 points each)
  - Problem solving (you must also to be able to solve new problems)
    - 2 problems (~ 10 points each)
- See Moodle for some examples (last year first exams and generic exercises)

13.1 ALTERNATIVE 2-APPROX ALGORITHM FOR VERTEX COVER

Exercise

Consider the following algorithm for Vertex Cover:

- *run DFS from an arbitrary vertex of G*
- *return all the non-leaf vertices of the DFS tree*

1) Show that this algorithm returns a Vertex Cover

2) Show that this algorithm is a 2-approx algorithm for Vertex Cover (Hint: show a large enough matching in the DFS tree)

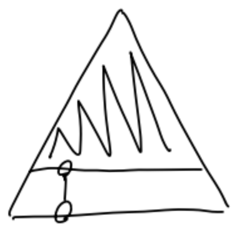
3) Show a lower bound of 2 to the approximation factor of this algorithm

Solution

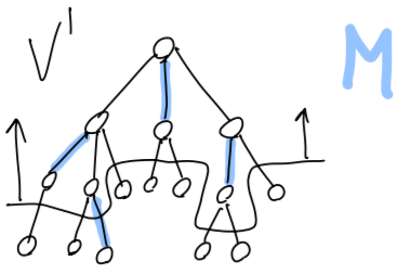
(Obviously, every vertex is covered. The question is: "Am I really covering all of the edges?". Basically, no edge can be in between of the two leaves)

1) The parents of the leaves cover all the edges left uncovered by the leaves of the DFS tree.

2) Let's help us with a picture and depict the DFS tree like the following. Let's try to get some large cover V'. We try to find some matching M taking all the vertices I can possibly get.



This is done level by level, adding as many edges as possible. This allows to construct a relatively large matching inside of the DFS tree. The matching can be considered maximal since it cannot be extended.



Let r be the root, choose one child v and add (r, u) to the matching M. Then, for every level $i \geq 1$ consider all vertices v not endpoints of any edge of M; choose a child u and add (v, u) to M. Then, repeat this process up to the leaves.

a) Upper bound to the cost of V' (which is our solution, greedy choice made by us)

By construction, M matches all the vertices of V' and since each of such edges has at most 2 endpoints in V', we can write:

$$|V'| \leq 2|M|$$

b) Lower bound to the cost of $V^* = OPT$ (which is the optimal solution, selected by the algorithm: VC with min amount of vertices)

∀ matching M of G, $|V^*| \geq |M|$ (size of V^* is at least the size of M).

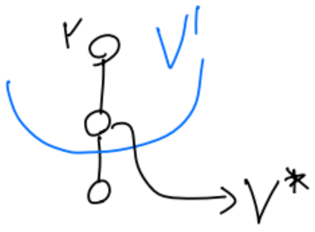
(this happens because if M is a matching $\Rightarrow$ in any vertex cover, in particular V^* there must be ≥ 1 vertex ∀ edge of M – seen before in this course)

Putting these inequalities together, we have:

$$\Rightarrow |V'| \leq \cancel{2|M|} \leq 2|V^*|$$

3) Show that the 2-factor is tight.

Consider a graph of 3 vertices. You run the DFS from the top vertex. Then, V' is the one which chooses two vertices, but OPT (V^*) is one.



A more general example is the *star graph*, having DFS starting from a leaf.

13.2 Single-linkage Clustering

We define the *clustering* as follows: given a set X of n data points, partition them into "coherent groups" (called clusters) of "similar points" (basically, subsets of similar points).

We define a similarity function$f$ as follows: it assigns to each pairs of data points a real number that specifies their "similarity" (takes in input two points and tells how similar they are). Consider the following: the smallest f = most similar points.

Goal: a k —clustering = partition data points into k non-empty clusters.

Single-linkage clustering: at the beginning, every data point is in its own cluster; then, successively merge the two clusters containing the most similar pair of points belonging to different clusters, until k clusters remain.

Exercise: Give a fast implementation of single-linkage clustering.

Idea: Hey, this is Kruskal's algorithm (stopped early) – in class, people had the intuition of using Union-Find sets, so to merge progressively clusters using that. This is the right way to think about it.

1) Define a complete graph $G = (X, E)$ with a vertex set X and one edge $(x, y) \in E$ of weight $f(x, y)$ for each pair of vertices $x, y \in X$.

2) Run Kruskal's algorithm on G until the solution T contains $n - k$ edges (or, equivalently, until k connected components remain).

3) Compute the connected components of (X, T) and return the corresponding partition of X

Complexity: $O(n^2 \log(n))$

13.3 Maximal Matching

Now, I will consider here for notes completion the approximation algorithm that was done in other years (basically each year since 2020-2021 up to 2022-2023, also found in existing video-lessons).

Exercise (matching – maximal/maximum matching)

Given a graph $G = (V, E)$, recall that a matching $M \subseteq E$ is a subset of edges that do not share vertices. We want to compute a matching of maximum size (that is, containing as many edges as possible).

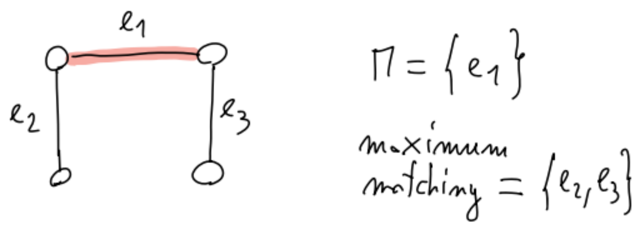
There exist polynomial-time algorithms, but those are slow/complicated. Consider the following simple algorithm:

```
$GREEDY\_MATCHING(G)$
    $m = |E|$
    $M = \emptyset$
    $let\ E = \{e_1, e_2, \dots, e_m\}$
    for $i = 1\ to\ m$ do:
        if $\forall e \in M \; e \cap e_i = \emptyset$ then
                $M = M \cup \{e_i\}$
        return $M$
```

Observation: this algorithm returns a maximal ($\neq$ maximum) matching → it can't be augmented

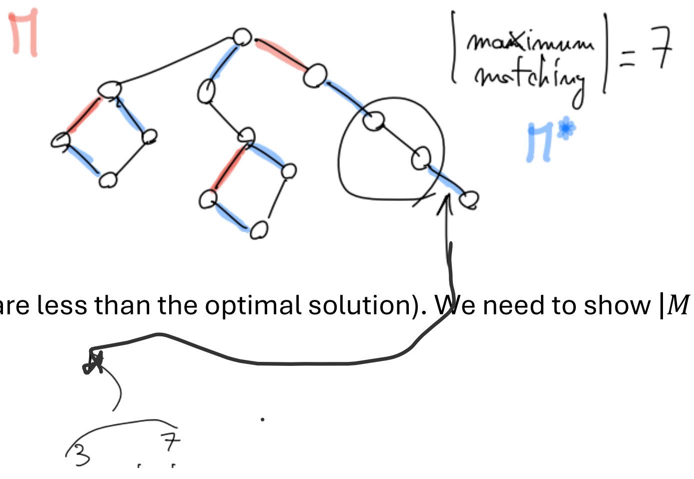
1) Give a graph G for which $GREEDY\_MATCHING$ returns a solution with half the edges of an optimal solution. The following is an example (build it as small as possible).

Basically, it's asking for a tight approximation, which is for the maximization problem. So, we'd have here $\frac{|M^*|}{|M'|} = 2$.



2) Prove that $GREEDY\_MATCHING$ is a 2-approximation algorithm (Hint: reason by contradiction – in past years when it was proposed in the Italian version of this course, it says "reason by contradiction starting from the hypothesis the matching returned by the algorithm has less the the half of the edges of the maximum cardinality).

Observe $GREEDY\_MATCHING$ returns a maximal matching: infact, $\forall e \in M, M \cup \{e\}$ is not a matching (otherwise, it would have been added to M). The algorithm continues to add more edges; the contradiction arises from the fact the algorithm has few edges and so the algorithm has not many edges to build a maximal matching and this is absurd, since is true by construction.


M
$\left|\begin{matrix}\text{maximum}\\\text{matching}\end{matrix}\right| = 7$
M^*
3
7

Clearly, $|M| \leq |M^*|$ (edges are less than the optimal solution). We need to show $|M| \geq \frac{|M^*|}{2}$.

Suppose, by contradiction, that $|M| < \frac{|M^*|}{2}$ edges. So, the edges of M cover at most $2|M| < |M^*|$ vertices $\Rightarrow \exists$ edge of M^* that does not cover any vertex covered by edges of M (without intersection with M edges) $\Rightarrow$ that edge(s) can be added to M, we call obtain a matching again: contradiction, given M is a maximal matching.

14 Exams and Older Exercises

In this section, all exams available (at the time of this writing) will be discussed with possible solutions + some additional exercises found within past editions of the course, which can be further helpful.

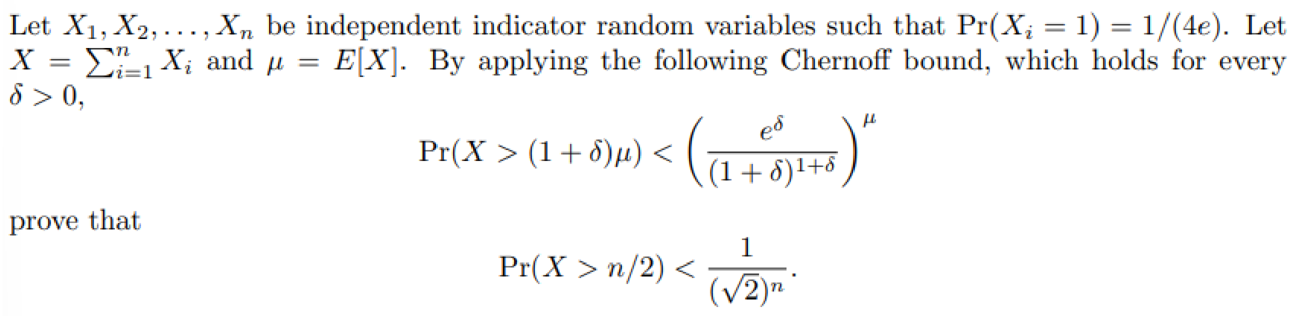
Honest advice: his "problem solving" part is more than enough covered knowing the theory and his exercises very well/perfectly since, as you will see, he will only ask variants of things done or showed exactly in that way in the theory like the question and keep an eye to hints. Do not look anywhere else for this exam and waste your time (actually, very few exercises of CLRS for Problem Solving part can be helpful/served as inspiration here), apart from existing exercises and exams.

14.1 Chernoff Bounds and How to use them

Chernoff Bounds are a set of powerful techniques used to provide tight bounds on the tail probabilities of sums of independent random variables.

- They are particularly useful for assessing the likelihood that the sum deviates significantly from its expected value
- Unlike simpler bounds like Markov's, Chernoff bounds take advantage of the distribution's specific characteristics to offer sharper estimates, especially useful for understanding the decay of tail probabilities exponentially fast

Consider the following footprint exercise:



Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that $\Pr(X_i = 1) = 1/(4e)$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. By applying the following Chernoff bound, which holds for every $\delta > 0$,

$$\Pr(X > (1+\delta)\mu) < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$$

prove that

$$\Pr(X > n/2) < \frac{1}{(\sqrt{2})^n}.$$

What is important in this set of exercises is the set of following steps (always like this):

- Characterize the event X_i (dependent on the type of problem you are dealing with)
  - And find the probability of success
- Characterize the expected value μ
- Use it to find δ
- Apply the bound given by the exercise

Consider we are usually bounding a precise value: apart from some *strange* cases, normally you have to get exactly the number given by the exercise.

So, here, we have to first consider X_i. We know they are independent. Now, we simply need to find the expected value. We already have here the probabilty of success given by $\Pr(X_i = 1) = \frac{1}{4e}$. This applies for all$n$ events since they are independent so:

$$\mu = E[X] = E[\sum_{i=1}^{n} E[X_i] = n * \frac{1}{4e} = \frac{n}{4e}$$

Now, we find δ and this is done according to value we have to bound, given by the exercise or explicitly told here like $\frac{n}{2}$:

$$(1+\delta)\mu = \frac{n}{2}$$

$$(1+\delta)\frac{n}{4e} = \frac{n}{2}$$

$$(1+\delta)\frac{n}{2e} = n$$

$$(1+\delta)n = 2e(n)$$

$$(1+\delta) = 2e$$

$$\delta = 2e - 1$$

Now that we found δ, let's plug it in back in the original bound:

$$\Pr(1+\delta)\,\mu < \left(\frac{(e^{\delta})}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

$$= \Pr\left(X > (1+2e-1)\frac{n}{4e}\right) \leq$$

$$\leq \left(\frac{e^{2e-1}}{(1+2e-1)^{(1+2e-1)}}\right)^{\frac{n}{4e}}$$

$$\leq \left(\frac{e^{2e-1}}{(2e)^{(2e)}}\right)^{\frac{n}{4e}}$$

$$\leq \left(\frac{\cancel{e^{2e}} * e^{-1}}{2^{2e} * (\cancel{e^{2e}})}\right)^{\frac{n}{4e}}$$

$$\leq \left(\frac{1}{e}\right)^{\frac{n}{4e}} * \left(\frac{1}{2^{2e}}\right)^{\frac{n}{4e}}$$

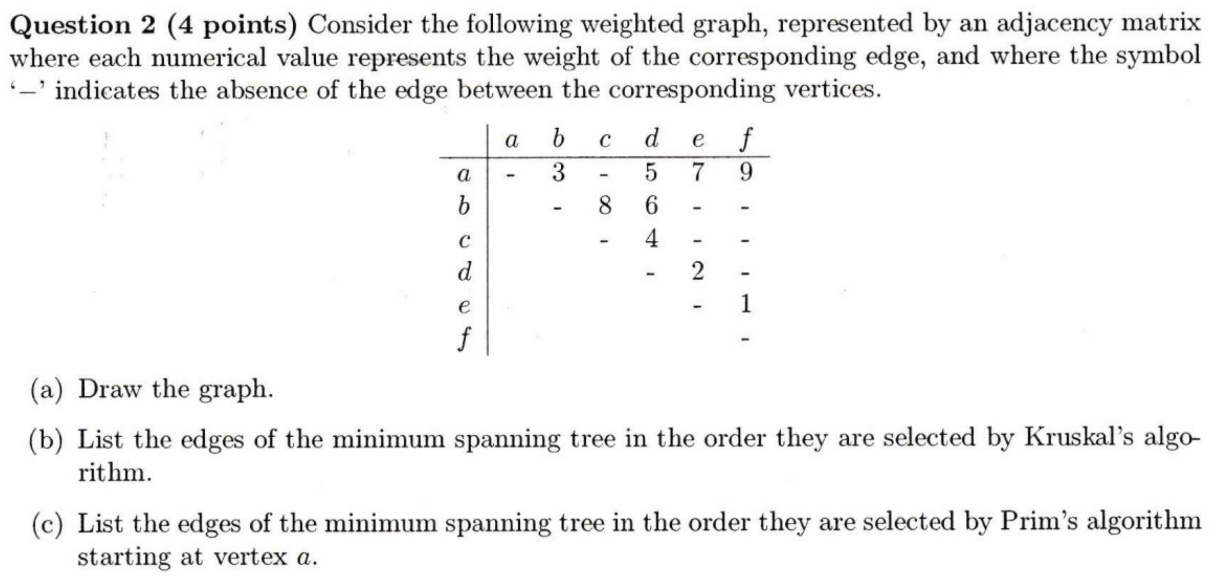
$$\leq \left(\frac{1}{e^{-4e}}\right)^{n} * \left(\frac{1}{2^{\frac{2e}{4e}}}\right)^{n}$$

$$\leq \left(\frac{1}{e^{-4e}}\right)^{n} * \left(\frac{1}{2^{\frac{1}{2}}}\right)^{n}$$

$$\leq \left(\frac{1}{e^{-4e}}\right)^{n} * \left(\frac{1}{\sqrt{2}}\right)^{n}$$

To infinity, it dominates the second factor, so we'd have $\left(\frac{1}{\sqrt{2}}\right)^{n}$

14.2 Exam of 04-09-2024



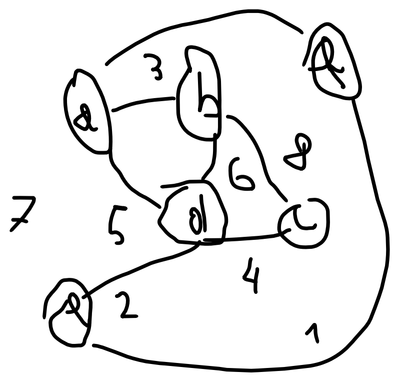
Question 2 (4 points) Consider the following weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding edge, and where the symbol '–' indicates the absence of the edge between the corresponding vertices.

	a	b	c	d	e	f
a	-	3	-	5	7	9
b		-	8	6	-	-
c			-	4	-	-
d				-	2	-
e					-	1
f						-

(a) Draw the graph.

(b) List the edges of the minimum spanning tree in the order they are selected by Kruskal's algorithm.

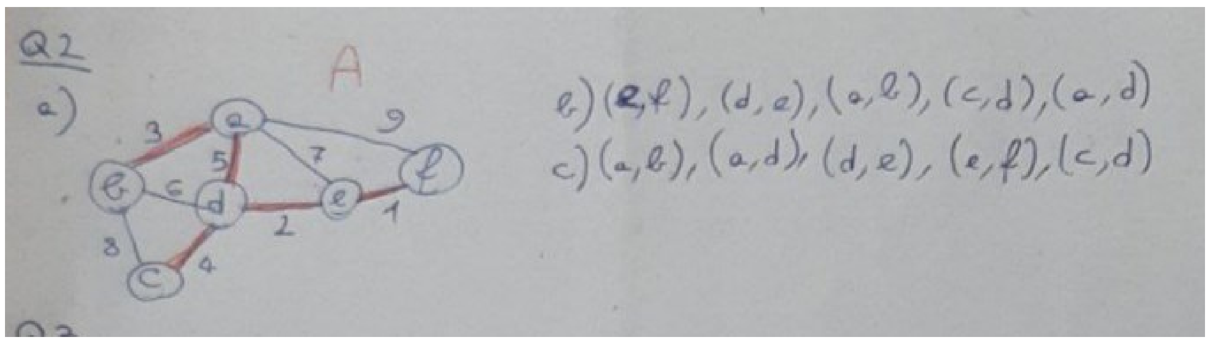
(c) List the edges of the minimum spanning tree in the order they are selected by Prim's algorithm starting at vertex a.



b) Kruskal: sort by weight in ascending order, then select edges: $\{e,f\},\{e,d\},\{a,b\},\{d,c\},\{a,d\}$

c) Prim: start from a source vertex then grow a spanning tree from there: $\{a,b\},\{a,d\},\{d,e\},\{e,f\},\{c,d\}$

Alternatively, a better graph here:



b) (e,f), (d,e), (a,b), (c,d), (a,d)
c) (a,b), (a,d), (d,e), (e,f), (c,d)

Question 3 (4 points) Define the set cover problem and briefly describe the $O(\log n)$-approximation algorithm seen in class.

Set cover is an optimization problem that models many problems requiring resources to be allocated. It aims to find the least number of subsets that cover some universal set.

Its inputs are:

- $I = (X, F)$ = instance of the set covering problem
- $X =$ set of elements of any kind, called "universe"
- $F \subseteq \{S : S \subseteq X\} = B(X)$
    a. B stands for "Boolean": set of all subsets of X

There is a constraint that needs to be always respected: $\forall x \in X, \exists\, S \in F : x \in S$ i.e., "F covers X"

Optimization problem: (smallest subset of F having its members covering all X) → find $F' \subseteq F$ s.t.

3) F' covers X
4) $\min\ |F'|$

Example:

$X = \{1,2,3,4,5\}$

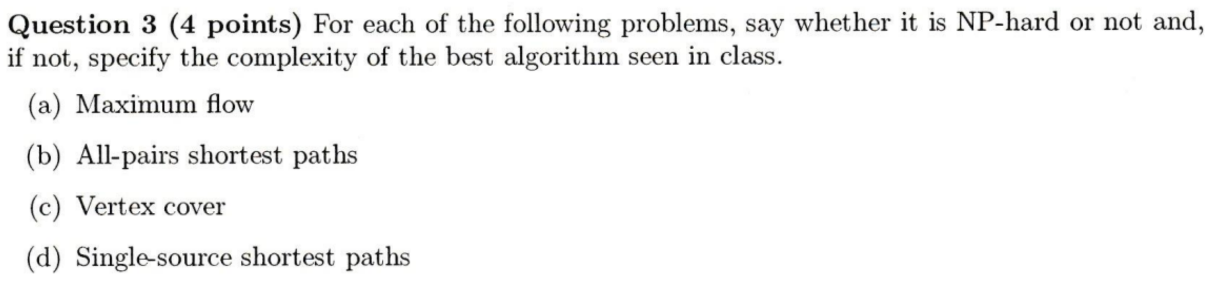
$F = \{\{1,2,3\}, \{2,4\}, \{3,4\}, \{4,5\}\}$

$\Rightarrow F^* = \{\{1,2,3\}, \{4,5\}\}$

The greedy method works by picking at each stage, the set S that covers the greatest number of remaining elements that are uncovered:

- choose the subset that contains the largest number of uncovered elements
- remove from X those covered elements
- repeat until $X = \emptyset$

```
$Approx\_Set\_Cover(X, F)$
    $U = X$
    $F' = \emptyset$ // solution
    while $U \neq \emptyset$: do
        // take the set of F covering as many elements as possible
        let $S \in F = |S \cap U| = \max_{S' \in F}\{|S' \cap U|\}$
        $U \leftarrow U \setminus S$      // update the list of available elements, removing those from S
        $F \leftarrow F \setminus \{S\}$    // remove the sets already considered inside of F
        $F' \leftarrow F' \cup \{S\}$
    return $F'$
```



Question 3 (4 points) For each of the following problems, say whether it is NP-hard or not and, if not, specify the complexity of the best algorithm seen in class.

(a) Maximum flow

(b) All-pairs shortest paths

(c) Vertex cover

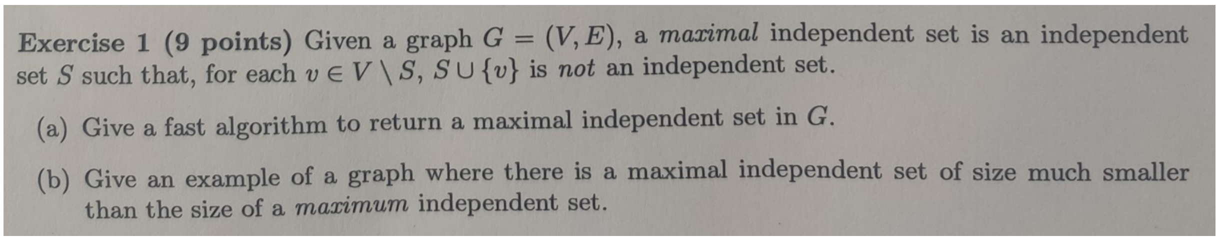
(d) Single-source shortest paths

(a) $O(m|f^*|)$

(b) $O(n^3)$

(c) NP-hard

(d) $O(n * m)$



Exercise 1 (9 points) Given a graph $G = (V, E)$, a *maximal* independent set is an independent set S such that, for each $v \in V \setminus S$, $S \cup \{v\}$ is *not* an independent set.

(a) Give a fast algorithm to return a maximal independent set in G.

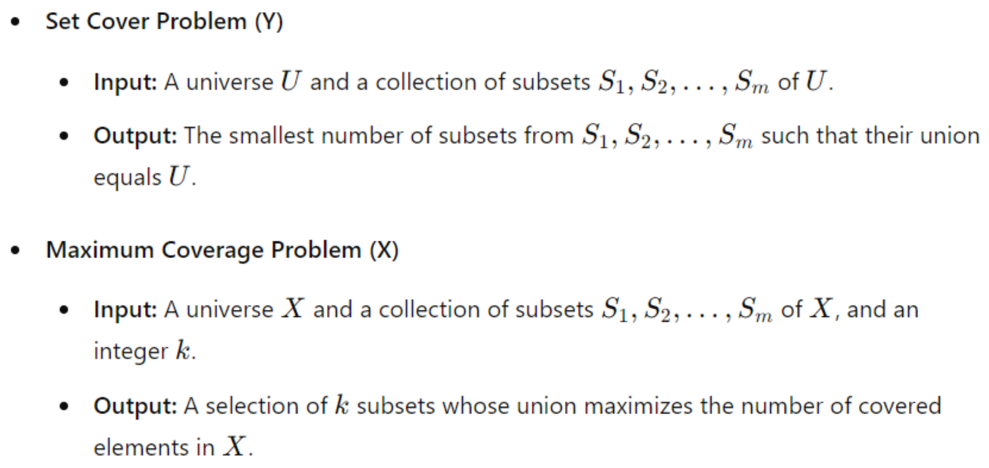
(b) Give an example of a graph where there is a maximal independent set of size much smaller than the size of a *maximum* independent set.

The most similar problem is Set Cover; given a universe U and a collection of subsets $S_1, S_2, \ldots S_n$ of U, the goal is to find the smallest number of subsets such that every element in U is covered by at least one of the selected subsets. A reference for this specific reduction can be also found here.

To do the reduction:

- Take an instance of Set Cover and use the same universe U with subsets $S_1, S_2, \ldots S_n$ for the maximum coverage problem

Specifically:



- **Set Cover Problem (Y)**
  - **Input:** A universe U and a collection of subsets $S_1, S_2, \ldots, S_m$ of U.
  - **Output:** The smallest number of subsets from $S_1, S_2, \ldots, S_m$ such that their union equals U.
- **Maximum Coverage Problem (X)**
  - **Input:** A universe X and a collection of subsets $S_1, S_2, \ldots, S_m$ of X, and an integer k.
  - **Output:** A selection of k subsets whose union maximizes the number of covered elements in X.

The reduction process would work as follows:

- Use the universe U from Set Cover as universe X for Maximum Coverage ($X = U$)
- For any given k representing the number of subsets, use the same subsets $S_1, S_2, \dots S_n$, this is the size of the optimal solution for Set Cover and Maximum Coverage alike
  - If Maximum Coverage can be solved in poly-time for any k, by trying all possible k from 1 to m, we can determine the smallest k such that the maximum coverage is $|U|$
- Evaluate Maximum Coverage for all possible values of k from 1 to m

To prove this is correct:

- if Set Cover has a solution for some k, it means the k subsets cover the entire universe U and applying this to Maximum Coverage will yield full coverage of U for the selected k, showing the maximum coverage is $|U|$; infact, this ensures the optimal selection of the correct number of subsets so to achieve the correct solution with respect to the number of elements of Coverage, ensuring a good enough cover for size k

- if Maximum Coverage can be solved efficiently for any k, this means we can iterate through the subsets $k = 1$ to m (where m is the number of subsets). The smallest k that results in Maximum Coverage equaling $|U|$ corresponds to the smallest k solving Set Cover

  a. Since the problem itself is NP-Hard, the ability to solve it using MCP implies that MCP must also be NP-hard, since we can check and evaluate Maximum Coverage for each subset k up to m and this happens since $\forall k$ up to m we check if the coverage equals $|U|$ and take the smallest one

This proof clearly shows how a polynomial-time solution to Maximum Coverage would enable a polynomial-time solution to Set Cover, establishing the NP-hardness of Maximum Coverage through reduction defined precisely.

Exercise 2 (10 points) - (not actually sure on the text, this was reconstructed)

Part 1:

Given a randomized Monte Carlo problem with complexity in the worst-case T(n) and failure probability $\epsilon \in (0,1)$, and a deterministic algorithm to check the correctness of the solution in constant time O(1), transform the Monte Carlo algorithm into a Las Vegas algorithm.

Part 2:

Prove that the complexity of the resulting Las Vegas algorithm is T(n) in the worst case.

1)

Given:

- A randomized Monte Carlo algorithm with worst-case complexity T(n) and failure probability $\epsilon \in (0,1)$.

- A deterministic algorithm for verifying the correctness of the solution in constant time O(1).

To transform the Monte Carlo algorithm into a Las Vegas algorithm:

1. Run the Monte Carlo algorithm to obtain a solution.

2. Use the deterministic verification algorithm to check the correctness of the solution.

   - If the solution is correct, return the solution and terminate the algorithm.

   - If the solution is incorrect, go back to step 1 and repeat the process.

The resulting algorithm is a Las Vegas algorithm because it always returns a correct solution, albeit with a variable running time.

2)

Let's analyze the expected running time of the Las Vegas algorithm:

- The probability of the Monte Carlo algorithm succeeding in a single iteration is $(1 - \epsilon)$.

- The expected number of iterations until a correct solution is found is $1 / (1 - \epsilon)$.

  - This is because the number of iterations follows a geometric distribution with success probability $(1 - \epsilon)$.

Now, let's calculate the expected running time:

- Each iteration of the Las Vegas algorithm consists of:

  - Running the Monte Carlo algorithm, which takes T(n) time.

  - Verifying the solution, which takes O(1) time.

- The expected running time is the product of the expected number of iterations and the time per iteration:

  - Expected running time = $(1 / (1 - \epsilon)) * (T(n) + O(1))$

  - Simplifying the expression, we get:

    - Expected running time = $(1 / (1 - \epsilon)) * T(n) + (1 / (1 - \epsilon)) * O(1)$

Since ϵ is a constant (independent of n), $(1 / (1 - \epsilon))$ is also a constant. Therefore:

- $(1 / (1 - \epsilon)) * T(n) = \Theta(T(n))$

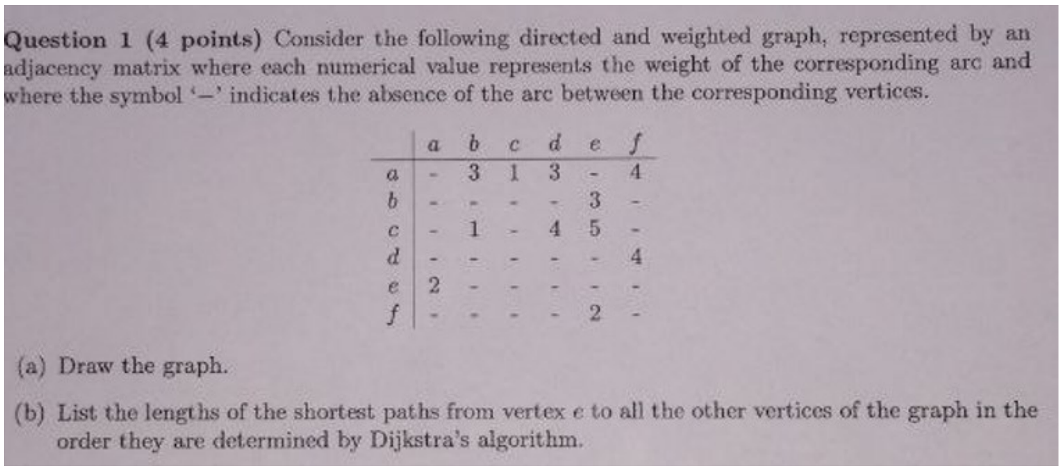
- $(1 / (1 - \epsilon)) * O(1) = O(1)$

The expected running time of the Las Vegas algorithm is $\Theta(T(n)) + O(1) = \Theta(T(n))$.

Thus, we have proven that the complexity of the resulting Las Vegas algorithm is T(n), the same as the worst-case complexity of the original Monte Carlo algorithm.

Written by Gabriel R.

14.3 Exam of 21-08-2024



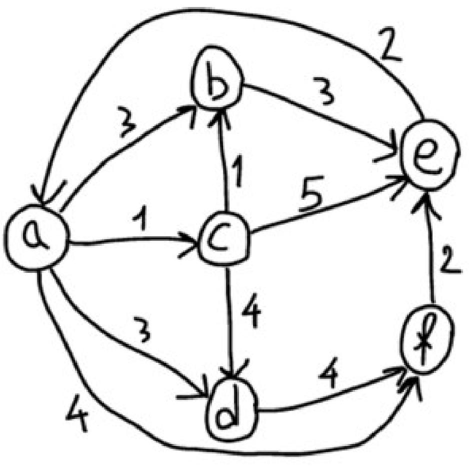
Question 1 (4 points) Consider the following directed and weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding arc and where the symbol '−' indicates the absence of the arc between the corresponding vertices.

	a	b	c	d	e	f
a	-	3	1	3	-	4
b	-	-	-	-	3	-
c	-	1	-	4	5	-
d	-	-	-	-	-	4
e	2	-	-	-	-	-
f	-	-	-	-	2	-

(a) Draw the graph.

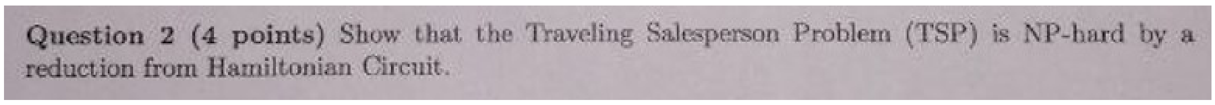
(b) List the lengths of the shortest paths from vertex e to all the other vertices of the graph in the order they are determined by Dijkstra's algorithm.

1.



2. $a - c : 1, a - b : 2, a - d : 3, a - f : 4, a - e : 5.$

3. $O((m + n) \log n)$.



Question 2 (4 points) Show that the Traveling Salesperson Problem (TSP) is NP-hard by a reduction from Hamiltonian Circuit.

Problem Definitions:

- Hamiltonian Circuit (HC):
    - Input: An undirected graph G = (V, E)
    - Question: Does G contain a cycle that visits each vertex exactly once?
- Traveling Salesperson Problem (TSP):
    - Input: A complete weighted graph G' = (V', E') and a target cost k
    - Question: Is there a tour (cycle visiting all vertices once) with total cost ≤ k?

The reduction would work this way: Given an instance of HC with graph G = (V, E), we construct an instance of TSP as follows: a) Let V' = V (same set of vertices) b) Create a complete graph G' = (V', E') where:

- If (u, v) ∈ E, set weight w(u, v) = 1
- If (u, v) ∉ E, set weight w(u, v) = 2 c) Set the target cost k = |V|

For the proof, we need to show that G has a Hamiltonian circuit if and only if G' has a tour with cost ≤ |V|. (⇒) If G has a Hamiltonian circuit:

- This circuit uses |V| edges from E
- In G', these edges all have weight 1
- Therefore, there's a tour in G' with total cost |V|

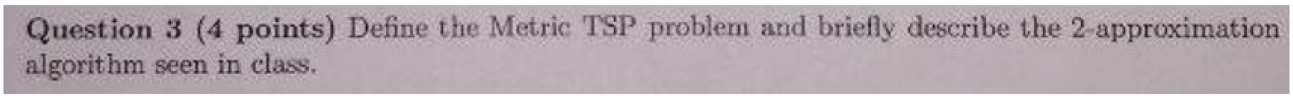
(⇐) If G' has a tour with cost ≤ |V|:

- The tour must use exactly |V| edges (as it visits all vertices once)
- For the cost to be ≤ |V|, all edges must have weight 1
- Edges with weight 1 in G' correspond to edges in G
- Therefore, this tour corresponds to a Hamiltonian circuit in G

This is done in polynomial-time as reduction, since:

- Creating G' requires checking each pair of vertices once: O(|V|^2)
- Setting edge weights is constant time per edge: O(1)
- Total time complexity: O(|V|^2)

In conclusion: Since HC is NP-complete, and we've shown a polynomial-time reduction from HC to TSP, we can conclude that TSP is NP-hard.



Question 3 (4 points) Define the Metric TSP problem and briefly describe the 2-approximation algorithm seen in class.

The Metric Traveling Salesperson Problem (Metric TSP) is a special case of the general TSP where the distances between cities satisfy the triangle inequality. Formally:

- Input: A complete, undirected graph G = (V, E) where V is the set of vertices (cities) and E is the set of edges (connections between cities). Each edge (u, v) has a non-negative weight w(u, v) representing the distance between cities u and v.

- The weights satisfy the triangle inequality: For any three vertices u, v, and w, w(u, v) ≤ w(u, w) + w(w, v).

- Goal: Find the shortest tour that visits each city exactly once and returns to the starting city.

2-Approximation Algorithm:

The 2-approximation algorithm for Metric TSP seen in class is as follows:

1. Compute a Minimum Spanning Tree (MST) T of the graph G.

2. Perform a depth-first search (DFS) traversal of T, listing vertices in the order they are visited.

3. Construct a Hamiltonian cycle H by following the order of vertices from the DFS traversal, but skipping any vertices that have already been visited.

4. Return H as the approximate solution.

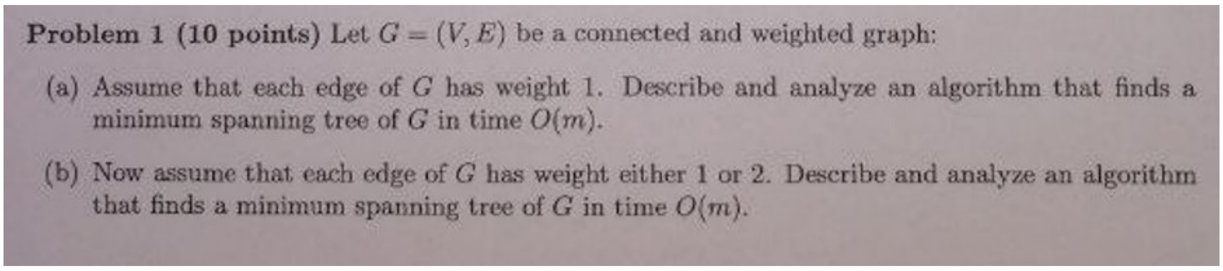
This algorithm guarantees a tour with total weight at most twice the weight of the optimal tour. The approximation factor of 2 is derived from:

- The weight of the MST is a lower bound on the optimal TSP tour.

- The DFS traversal of the MST visits each edge twice.

- Shortcutting (skipping repeated vertices) can only decrease the tour length due to the triangle inequality.

This algorithm is efficient, running in O(n^2) time for a graph with n vertices, as computing the MST and performing DFS can both be done in O(n^2) time for a complete graph.



Problem 1 (10 points) Let $G = (V, E)$ be a connected and weighted graph:

(a) Assume that each edge of G has weight 1. Describe and analyze an algorithm that finds a minimum spanning tree of G in time $O(m)$.

(b) Now assume that each edge of G has weight either 1 or 2. Describe and analyze an algorithm that finds a minimum spanning tree of G in time $O(m)$.

To solve this problem, we reference minimum spanning tree algorithms, particularly Kruskal's algorithm and its efficient implementation using the Union-Find data structure.

(a) When all edges have weight 1:

In this case, any spanning tree will be a minimum spanning tree, since all spanning trees will have the same total weight of n-1 (where n is the number of vertices). We can use a simple breadth-first search (BFS) or depth-first search (DFS) to find a spanning tree in O(m) time, where m is the number of edges.

Algorithm:

1. Choose an arbitrary starting vertex s.

2. Perform a BFS or DFS from s, keeping track of the edges used to discover new vertices.

3. The set of edges used forms a minimum spanning tree.

Analysis:

- Time complexity: O(m) - we visit each edge at most once.

- Space complexity: O(n) - for the queue or stack used in BFS/DFS.

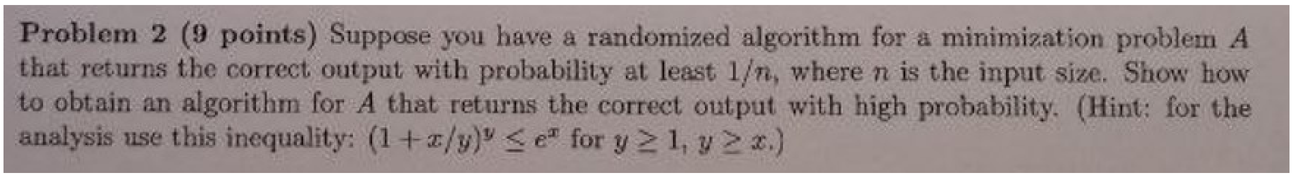
(b) When edges have weights 1 or 2:

For this case, we can modify Kruskal's algorithm to run in O(m) time by exploiting the fact that there are only two possible edge weights.

Algorithm:

```
1. Partition the edges into two sets: E1 (edges with weight 1) and E2 (edges with weight 2).
2. Initialize a Union-Find data structure for the vertices.
3. Process all edges in E1 first:
  - For each edge (u,v) in E1:
    - If Find(u) ≠ Find(v):
      - Add (u,v) to the MST
      - Union(u,v)
4. If the MST is not complete, process edges in E2:
  - For each edge (u,v) in E2:
    - If Find(u) ≠ Find(v):
      - Add (u,v) to the MST
      - Union(u,v)
```

This algorithm ensures we use as many weight-1 edges as possible before using any weight-2 edges, thus guaranteeing a minimum spanning tree. The O(m) time complexity is achieved by avoiding the need to sort edges, which is typically required in Kruskal's algorithm.



Problem 2 (9 points) Suppose you have a randomized algorithm for a minimization problem A that returns the correct output with probability at least $1/n$, where n is the input size. Show how to obtain an algorithm for A that returns the correct output with high probability. (Hint: for the analysis use this inequality: $(1 + x/y)^y \leq e^x$ for $y \geq 1$, $y \geq x$.)

Here, we use Karger since the hint uses that inequality and the only point in the program we saw that is exactly that analysis – so, that's why we use that in place of a "normal" Chernoff Bound.

Characterize the event of getting a probability *at least* $\frac{1}{n}$ using Karger's analysis; run different times the analysis and fix a constant d, $d > 0$:

$$\Pr\left(X > \frac{1}{n}\right) > \frac{1}{n^d}$$

Since the probability is at least $\frac{1}{n}$, so we characterize using Karger. Here, we will characterize the probability of failure (the complement with respect to the previous, so $1 - \frac{1}{n}$, running k times to reduce the error probability.

We want to find a value for k such that $Pr\left(1 - \frac{1}{n}\right)^k \leq \frac{1}{n^d}$. In this case, it's standard the use of this inequality:

$$\left(1 + \frac{x}{y}\right)^y \leq e^x, y \geq 1, y \geq x$$

This inequality is derived from the exponential function and the binomial expansion. It represents an upper bound on the expression $\left(1+\frac{x}{y}\right)^y$, showing that it grows slower than e^x.

Now, we use Karger's analysis, in place of using $k = n^2$ we use $k = n$ and everything comes naturally.

By choosing $k = dn\ln(n)$ it follows that (it holds $d = 1$ coming from Karger):

$$\left(1-\frac{1}{n}\right)^{k=n} \leq e^{-1} = \frac{1}{e} \rightarrow \text{is } \underline{\text{not}} \text{ in the form } \frac{1}{n^d}$$

Recall the following from the Karger analysis (choice of k and rest of reasoning is that):

$$e^{-\ln(n^d)} = \frac{1}{n^d}$$

Continuing using Karger and the inequality:

$$\left(\left(1-\frac{1}{n}\right)^n\right)^{\ln(n^d)} = \left(1-\frac{1}{n}\right)^{n\ln(n)}$$

Let's wrap up:

$$\left(1-\frac{1}{n}\right)^{k=n\ln(n)} = \left(\left(1-\frac{1}{n}\right)^n\right)^{\ln(n)}$$

$$\leq (e^{-1})^{\ln(n)} = e^{-\ln(n)} = \frac{1}{n}$$

<u>For reference from theory, Karger's analysis is this.</u>

$\Pr(\textit{the k runs of FULL\_CONTRACTION do not return the min cut}) \leq \left(1-\frac{2}{n^2}\right)^k \leq \frac{1}{n^d}$ for some constant $d > 0$

The previous one is the probability of an unsuccessful event, so we want it very low, something like $\frac{1}{n^d}$.

We want to find a value for k such that $\left(1-\frac{2}{n^2}\right)^k \leq \frac{1}{n^d}$. In this case, it's standard the use of this inequality:

$$\left(1+\frac{x}{y}\right)^y \leq e^x, y \geq 1, y \geq x$$

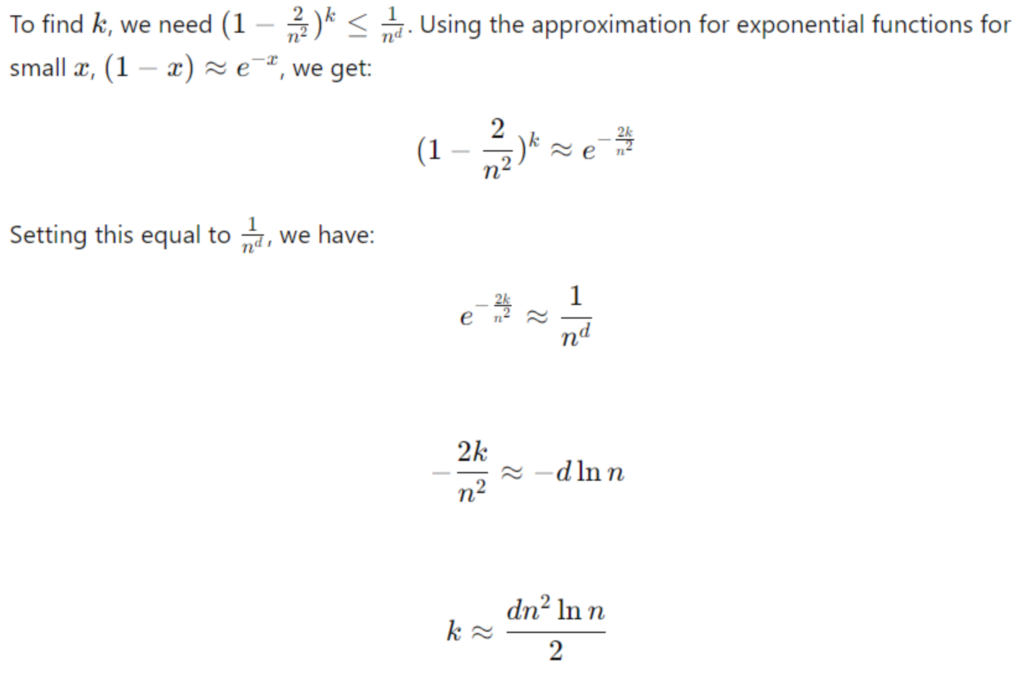
This inequality is derived from the exponential function and the binomial expansion. It represents an upper bound on the expression $\left(1+\frac{x}{y}\right)^y$, showing that it grows slower than e^x. The probability of not contracting the minimum cut in each iteration needs to be bounded and manipulated to ensure the overall algorithm's success probability is high.

By choosing $k = \frac{dn^2\ln(n)}{2}$ it follows that:

$$\left(\left(1-\frac{2}{n^2}\right)^{n^2}\right)^{\ln(n^d)} \leq e^{-\ln(n^d)} = \frac{1}{n^d}$$

Given I am curious, I asked myself: why exactly that value for k?

Consider the probability of success if $\frac{2}{n^2}$ while the failure is, by complement, $1-\frac{2}{n^2}$ which, amplified by k runs, becomes $\left(1-\frac{2}{n^2}\right)^k$. The constant d is the desired level of confidence to keep the wanted threshold (in this case $\frac{1}{n^d}$) as low as possible. Then, using some good old GPT-4:



To find k, we need $(1-\frac{2}{n^2})^k \leq \frac{1}{n^d}$. Using the approximation for exponential functions for small x, $(1-x) \approx e^{-x}$, we get:

$$(1-\frac{2}{n^2})^k \approx e^{-\frac{2k}{n^2}}$$

Setting this equal to $\frac{1}{n^d}$, we have:

$$e^{-\frac{2k}{n^2}} \approx \frac{1}{n^d}$$

$$-\frac{2k}{n^2} \approx -d\ln n$$

$$k \approx \frac{dn^2 \ln n}{2}$$

Moving on:

$$\left(1-\frac{2}{n^2}\right)^{k=n^2} \leq e^{-2} = \frac{1}{e^2} \rightarrow \text{is } \underline{\text{not}} \text{ in the form } \frac{1}{n^d}$$

Recall the following:

$$e^{-\ln(n^d)} = \frac{1}{n^d}$$

Let's apply that:

$$\left(\left(1-\frac{2}{n^2}\right)^{n^2}\right)^{\ln(n^d)} = \left(1-\frac{2}{n^2}\right)^{n^2\ln(n^d)}$$

Let's wrap up (here, in the prof. notes, d magically disappears, but I assume it to be 1 so this works):

$$\left(1-\frac{2}{n^2}\right)^{\boxed{k=\frac{dn^2\ln(n^d)}{2}}} = \left(\left(1-\frac{2}{n^2}\right)^{n^2}\right)^{\frac{\ln(n^d)}{2}}$$

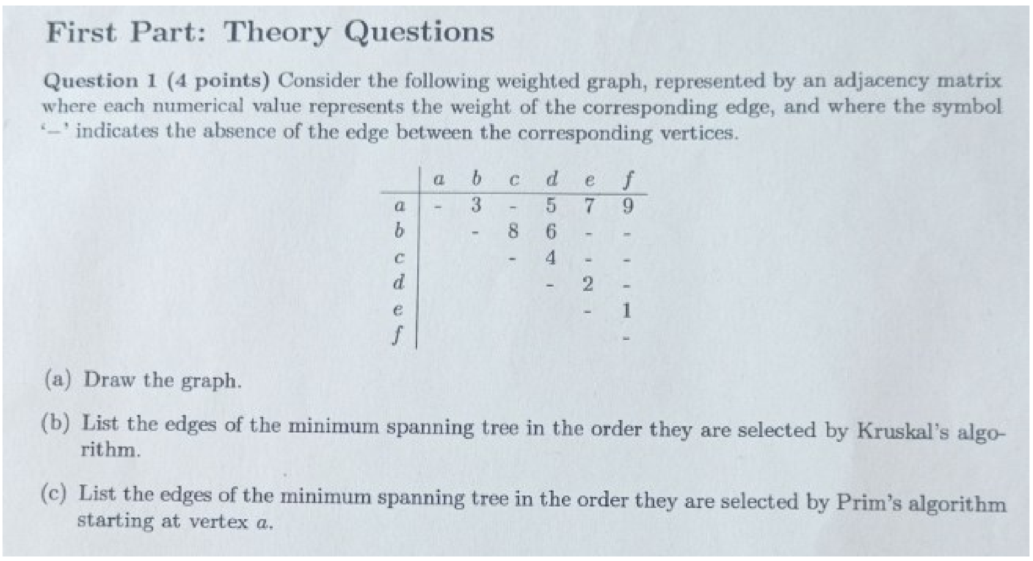
$$\leq (e^{-2})^{\frac{\ln(n^d)}{2}} = e^{-\ln(n^d)} = n^d = \frac{1}{n^d}$$

Then, by choosing that value for k the Karger's algorithm succeeds with high probability:

$$\Rightarrow \Pr(KARGER\ succeeds) > 1-\frac{1}{n^d}$$

So, in the end, the Karger algorithm accumulates the size of the min-cut with probability at least $\frac{1}{n^d}$.

14.4 Exam of 08-07-2024



First Part: Theory Questions

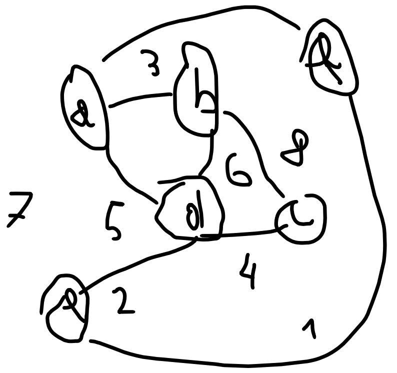
Question 1 (4 points) Consider the following weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding edge, and where the symbol '–' indicates the absence of the edge between the corresponding vertices.

	a	b	c	d	e	f
a	-	3	-	5	7	9
b		-	8	6	-	-
c			-	4	-	-
d				-	2	-
e					-	1
f						-

(a) Draw the graph.

(b) List the edges of the minimum spanning tree in the order they are selected by Kruskal's algorithm.

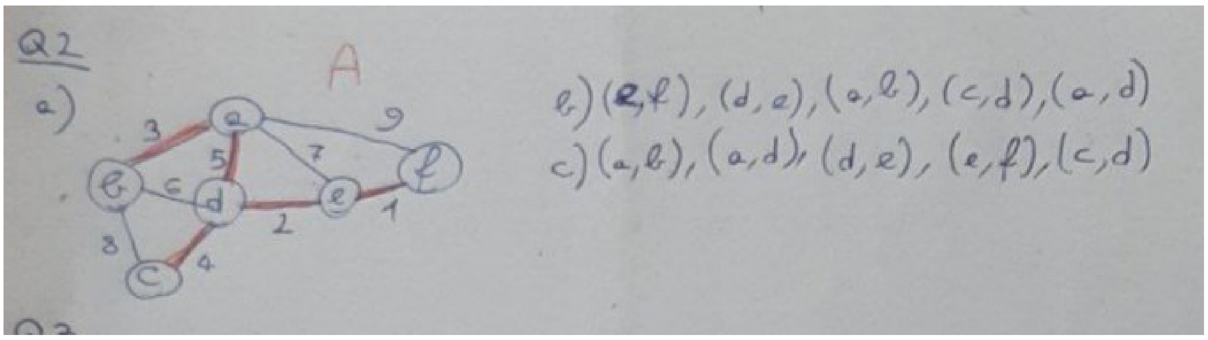
(c) List the edges of the minimum spanning tree in the order they are selected by Prim's algorithm starting at vertex a.



b) Kruskal: sort by weight in ascending order, then select edges: $\{e,f\},\{e,d\},\{a,b\},\{d,c\},\{a,d\}$

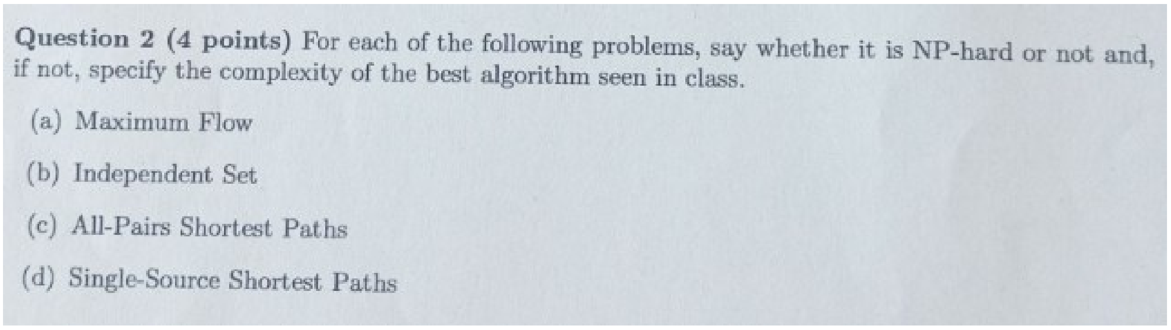
c) Prim: start from a source vertex then grow a spanning tree from there: $\{a,b\},\{a,d\},\{d,e\},\{e,f\},\{c,d\}$

Alternatively, a better graph here:



b) (e,f), (d,e), (a,b), (c,d), (a,d)

c) (a,b), (a,d), (d,e), (e,f), (c,d)



Question 2 (4 points) For each of the following problems, say whether it is NP-hard or not and, if not, specify the complexity of the best algorithm seen in class.

(a) Maximum Flow

(b) Independent Set

(c) All-Pairs Shortest Paths

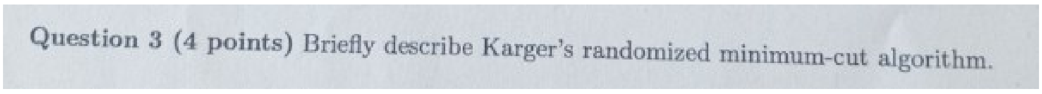
(d) Single-Source Shortest Paths

(a) $O(m|f^*|)$

(b) $O(n^3)$

(c) NP-hard

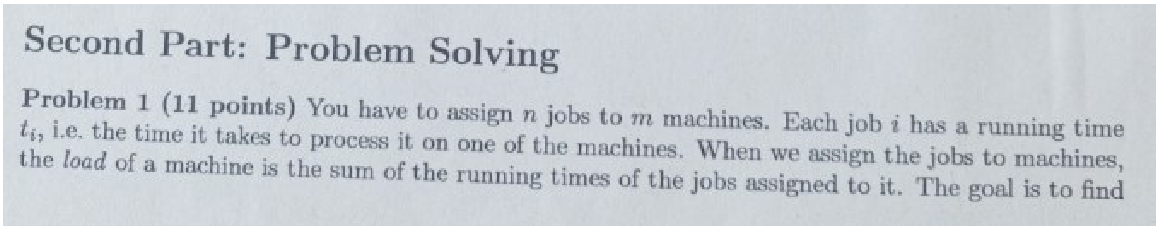
(d) $O(n * m)$



Question 3 (4 points) Briefly describe Karger's randomized minimum-cut algorithm.

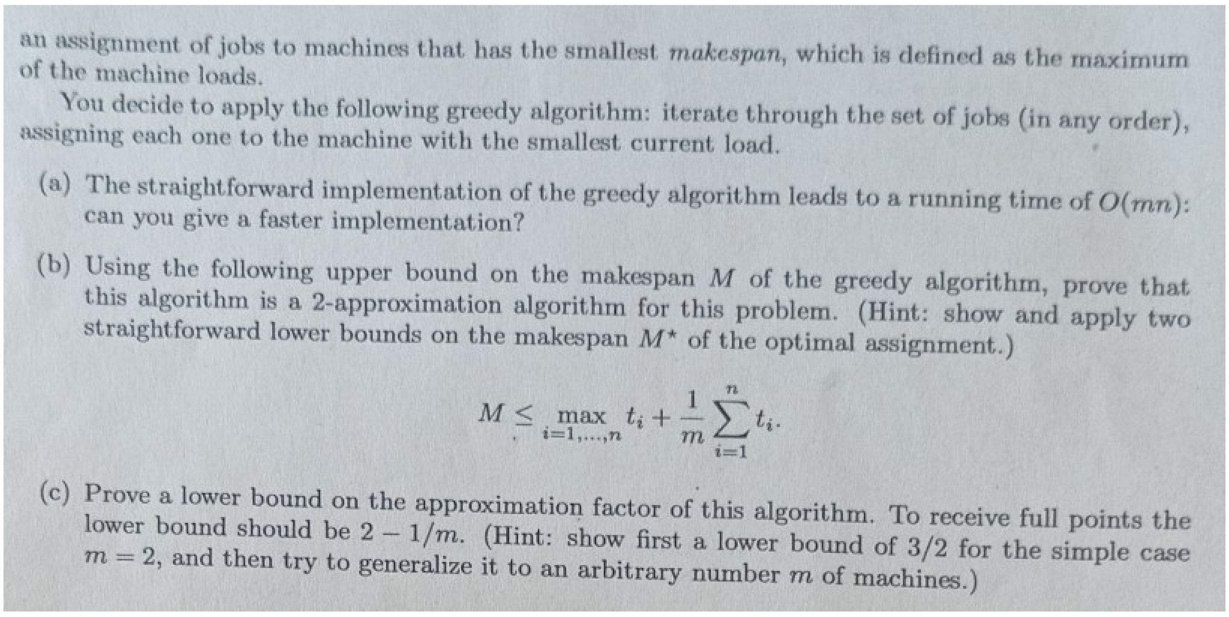
Karger is a Monte Carlo (randomized algorithm which may fail) algorithm which considers and solves the problem of the minimum cut (cut of minimum size so to remove the minimum number of edges to make the graph disconnected). It revolves around the concept of "contraction" (have two vertices merge into each other), so choosing an edge at random, contract the two vertices of that edge and removing all of the edges incident to both vertices, keeping the smallest cut found.

The algorithm is probabilistic, meaning it may not always find the true minimum cut, but repeating it increases the probability of finding the correct result. The number of repetitions needed for a high probability of success is typically polynomial in the number of vertices.



Second Part: Problem Solving

Problem 1 (11 points) You have to assign n jobs to m machines. Each job i has a running time t_i, i.e. the time it takes to process it on one of the machines. When we assign the jobs to machines, the *load* of a machine is the sum of the running times of the jobs assigned to it. The goal is to find


an assignment of jobs to machines that has the smallest *makespan*, which is defined as the maximum of the machine loads.

You decide to apply the following greedy algorithm: iterate through the set of jobs (in any order), assigning each one to the machine with the smallest current load.

(a) The straightforward implementation of the greedy algorithm leads to a running time of $O(mn)$: can you give a faster implementation?

(b) Using the following upper bound on the makespan M of the greedy algorithm, prove that this algorithm is a 2-approximation algorithm for this problem. (Hint: show and apply two straightforward lower bounds on the makespan M^* of the optimal assignment.)

$$M \leq \max_{i=1,\ldots,n} t_i + \frac{1}{m} \sum_{i=1}^{n} t_i.$$

(c) Prove a lower bound on the approximation factor of this algorithm. To receive full points the lower bound should be $2 - 1/m$. (Hint: show first a lower bound of 3/2 for the simple case $m = 2$, and then try to generalize it to an arbitrary number m of machines.)

(a)

Let's consider the steps involved in a straightforward implementation of the greedy algorithm:

1. Sort the jobs in non-increasing order of their running times. This takes $O(nlog(n)$ time.
2. Iterate through the sorted jobs and assign each job to the machine with the smallest current load. This step takes $O(n)$ time to iterate through the jobs, and for each job, we need to find the machine with the minimum load, which takes $O(m)$ time. Therefore, this step takes $O(nm)$

The total running time is $O(n \ log \ n) + O(nm)$. In the worst case, m could be O(n), making the running time $O(n^2)$. However, when m is considered a constant or bounded by a constant, the running time is dominated by $O(n \ log \ n)$.

Alternatively:

The straightforward implementation of the greedy algorithm leads to a running time of $O(mn)$ This is because for each job, you need to find the machine with the smallest load, which takes $O(m)$ time, and there are n jobs.

To achieve a faster implementation, we can use a min-heap (or priority queue) to keep track of the loads of the machines. The heap allows us to efficiently find and update the machine with the smallest load. These are the implementation steps:

1. Initialize a min-heap with the loads of all mmm machines (all initially 0).
2. For each job t_i:
    1. Extract the minimum load from the heap.
    2. Add t_i to this load.
    3. Insert the updated load back into the heap.

Using a min-heap, each insertion and extraction operation takes $O(\log(m))$ time, so the overall running time is $O(nlog(m))$.

(b)

To prove that the greedy algorithm is a 2-approximation algorithm, we need to show that for any input instance, the makespan M produced by the greedy algorithm is at most 2 times the optimal makespan M^*. Let's start with the given upper bound on the makespan M of the greedy algorithm:

$$M \leq \max_{i=1,\ldots n} t_i + \frac{1}{m}\sum_{i=1}^{n} t_i$$

We can derive two lower bounds on the optimal makespan M^*:

1. The optimal makespan M^* is at least the maximum running time of any job: $M^* \geq \max_{i=1,\ldots n} t_i$. This is because the job with the maximum running time must be assigned to some machine, and that machine will have a load of at least $\max_{i=1,\ldots n} t_i$.

2. The optimal makespan M^* is at least the average load on the machines: $M^* \geq \left(\frac{1}{m}\right)^* \sum_{i=1}^{n} t_i$ i=1 ti This is because the total load (sum of all job running times) must be distributed among the m machines, and the average load is a lower bound on the maximum load.

Using these two lower bounds, we can prove that $M \leq 2M^*$:

$$M \leq \max_{i=1,\ldots n} t_i + \frac{1}{m}\sum_{i=1}^{n} t_i \leq M^* + M^* = 2M^*$$

Therefore, the greedy algorithm is a 2-approximation algorithm for this problem.

To generalize this result to any number of machines m:

Let M be the makespan of the greedy algorithm, and M* be the optimal makespan. We have:

$$M \leq \max_{i=1,\ldots n} t_i + \frac{1}{m}\sum_{i=1}^{n} t_i \leq M^* + M^* = 2M^*$$

Thus, the greedy algorithm is a 2-approximation algorithm for the problem of minimizing the makespan when assigning n jobs to m machines.

Alternatively:

We need to show that the greedy algorithm provides a makespan M such that:

$M \leq \max_{i=1,...,n} t_i + \frac{1}{m} \sum_{i=1}^{n} t_i$

and then prove that the greedy algorithm is a 2-approximation.

1. **Upper Bound Proof**:
   - Let $T = \sum_{i=1}^{n} t_i$ be the total processing time of all jobs.
   - Let $t_{\max} = \max_{i=1,...,n} t_i$ be the maximum job time.
   - At any point in time during the execution of the greedy algorithm, let the load of the most loaded machine be L.

Since the greedy algorithm assigns each job to the machine with the least load:
$L \leq \frac{T}{m} + t_{\max}$

This inequality holds because, at the worst case, the last job (which is the largest job $t_{\max}$) can be added to the load of the most loaded machine before the job is assigned, resulting in an additional $t_{\max}$.

Thus,
$M \leq t_{\max} + \frac{T}{m}$

2. **2-Approximation Proof**:
   - Let M^* be the makespan of the optimal assignment.
   - A lower bound on M^* is $\frac{T}{m}$, because the total processing time T must be distributed among m machines, so at least one machine must have at least $\frac{T}{m}$ load.
   - Another lower bound on M^* is $t_{\max}$, since the largest job cannot be split and must be assigned to one machine.

Hence,
$M^* \geq \max\left(t_{\max}, \frac{T}{m}\right)$

From the previous inequality, we have:
$M \leq t_{\max} + \frac{T}{m}$
Given that:
$M^* \geq \max\left(t_{\max}, \frac{T}{m}\right)$
it follows that:
$t_{\max} + \frac{T}{m} \leq 2\max\left(t_{\max}, \frac{T}{m}\right)$

Thus, the makespan M of the greedy algorithm is at most twice the optimal makespan M^*, proving that the greedy algorithm is a 2-approximation algorithm.

(c)

To prove a lower bound on the approximation factor of the greedy algorithm, we need to find an instance where the ratio between the makespan produced by the greedy algorithm (M) and the optimal makespan (M*) is as large as possible. We will consider the simple case where m = 2 and the number of jobs n is arbitrary.

Consider the following instance:

- Job 1 has a running time of $t_1 = 1$
- Jobs 2 $to\ n$ have running times of $t_2 = t_3 = \cdots = t_n = \frac{1}{2}$

The optimal assignment for this instance is:

- Machine 1: Job 1
- Machine 2: Jobs 2 to n

This results in an optimal makespan of M* = 1.

Now, let's consider what the greedy algorithm might do. In the worst case, the greedy algorithm may assign jobs in the following way:

- Machine 1: Job 1, and half of the remaining jobs (i.e., $\frac{n-1}{2}$ jobs if n is odd, or $\frac{n-2}{2}$ jobs if n is even)
- Machine 2: The other half of the remaining jobs

In this case, the makespan of the greedy algorithm (M) will be:

$M = 1 + (\frac{n-1}{2} * \frac{1}{2})$ if n is odd

$M = 1 + (\frac{n-2}{2} * \frac{1}{2})$ if n is even

As n approaches infinity, the ratio $\frac{M}{M^*}$ approaches $\frac{3}{2}$:

$$\lim_{n\to\infty} \frac{1 + \left(\frac{n-1}{2} * \frac{1}{2}\right)}{1} = \frac{3}{2} \text{ for odd n}$$

$$\lim_{n\to\infty} \frac{1 + \left(\frac{n-2}{2} * \frac{1}{2}\right)}{1} = \frac{3}{2} \text{ for even n}$$

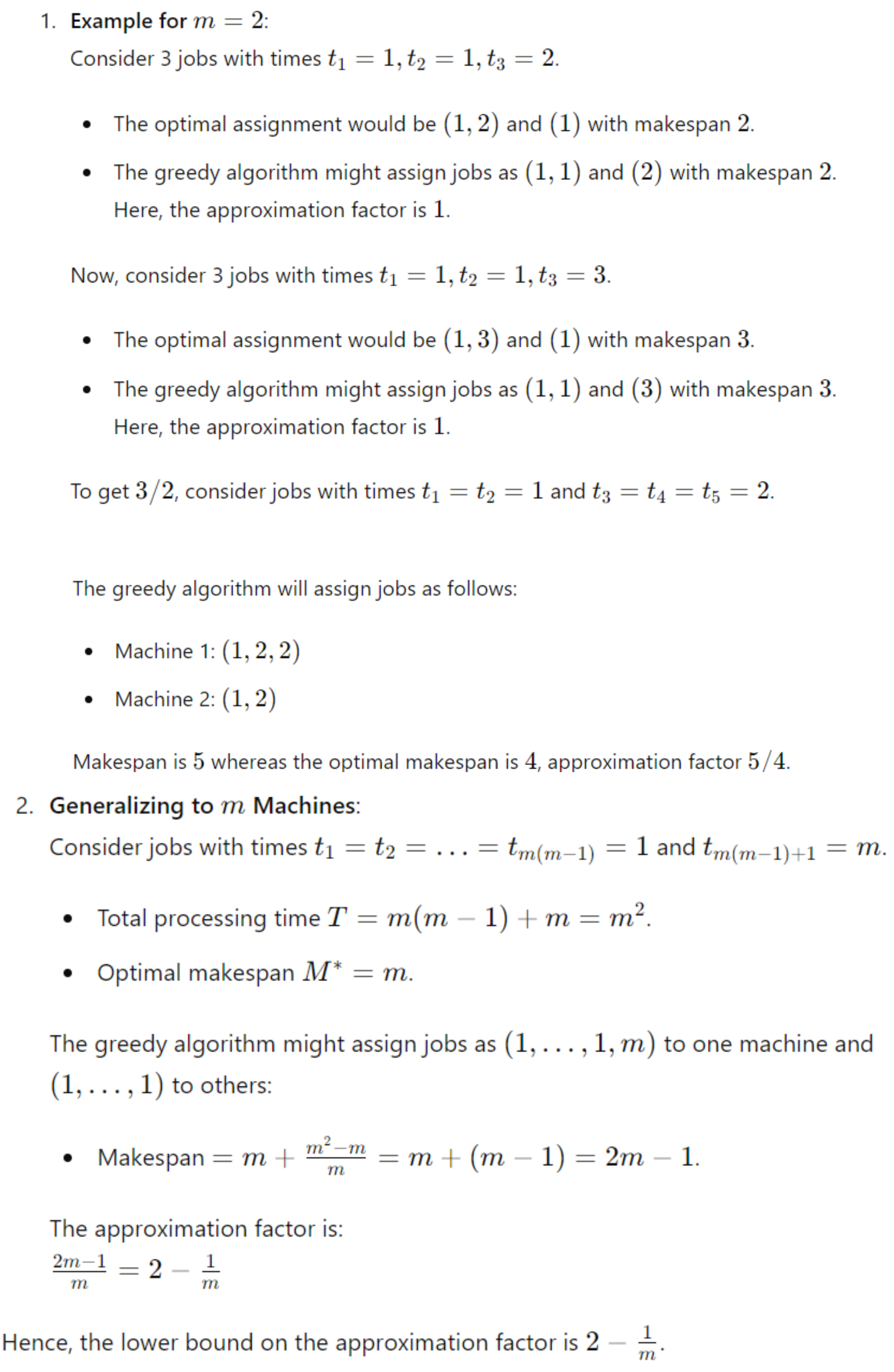
This shows that the approximation factor of the greedy algorithm can be as bad as $\frac{3}{2}$ in the simple case where $m = 2$ and n is arbitrary.

To prove that this lower bound of $\frac{3}{2}$ is tight for this case, we need to show that for any instance with $m = 2$ machines and n jobs, the ratio $\frac{M}{M^*}$ is always less than or equal to $\frac{3}{2}$.

From part (b), we know that for any instance, $M \leq 2M^*$ Therefore, $\frac{M}{M^*} \leq 2$

Thus, the lower bound of $\frac{3}{2}$ is tight for the case where $m = 2$ and n is arbitrary, as we have found an instance that achieves this ratio, and we have shown that the ratio cannot exceed 2 for any instance in this case.

Alternatively:



1. **Example for $m = 2$:**
   Consider 3 jobs with times $t_1 = 1, t_2 = 1, t_3 = 2$.

   - The optimal assignment would be $(1, 2)$ and (1) with makespan 2.
   - The greedy algorithm might assign jobs as $(1, 1)$ and (2) with makespan 2. Here, the approximation factor is 1.

   Now, consider 3 jobs with times $t_1 = 1, t_2 = 1, t_3 = 3$.

   - The optimal assignment would be $(1, 3)$ and (1) with makespan 3.
   - The greedy algorithm might assign jobs as $(1, 1)$ and (3) with makespan 3. Here, the approximation factor is 1.

   To get $3/2$, consider jobs with times $t_1 = t_2 = 1$ and $t_3 = t_4 = t_5 = 2$.

   The greedy algorithm will assign jobs as follows:

   - Machine 1: $(1, 2, 2)$
   - Machine 2: $(1, 2)$

   Makespan is 5 whereas the optimal makespan is 4, approximation factor $5/4$.

2. **Generalizing to m Machines:**
   Consider jobs with times $t_1 = t_2 = \ldots = t_{m(m-1)} = 1$ and $t_{m(m-1)+1} = m$.

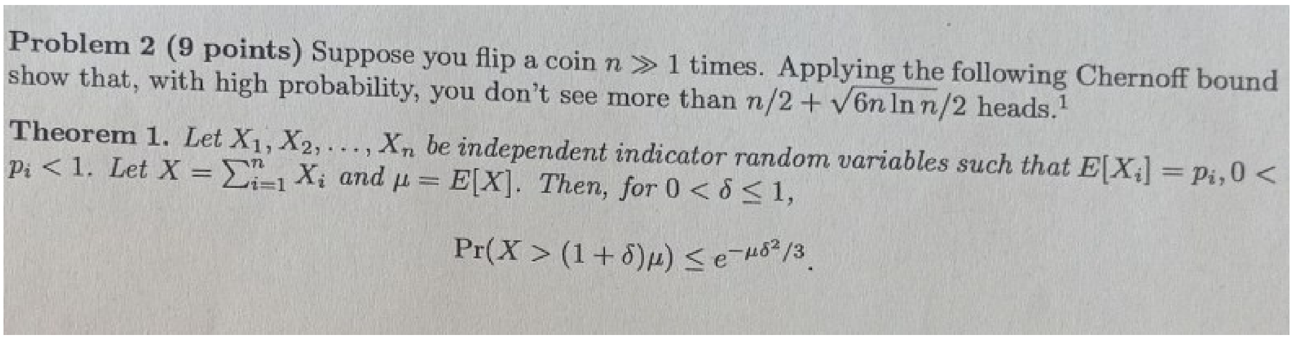
   - Total processing time $T = m(m-1) + m = m^2$.
   - Optimal makespan $M^* = m$.

   The greedy algorithm might assign jobs as $(1, \ldots, 1, m)$ to one machine and $(1, \ldots, 1)$ to others:

   - Makespan $= m + \frac{m^2 - m}{m} = m + (m - 1) = 2m - 1$.

   The approximation factor is:
   $\frac{2m-1}{m} = 2 - \frac{1}{m}$

Hence, the lower bound on the approximation factor is $2 - \frac{1}{m}$.


Problem 2 (9 points) Suppose you flip a coin $n \gg 1$ times. Applying the following Chernoff bound show that, with high probability, you don't see more than $n/2 + \sqrt{6n \ln n}/2$ heads.[1]

Theorem 1. *Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that $E[X_i] = p_i, 0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then, for $0 < \delta \leq 1$,*

$$\Pr(X > (1+\delta)\mu) \leq e^{-\mu\delta^2/3}.$$

First and foremost, $X_1 = 1$ if coin is tail, 0 otherwise. $X_i's$ are independent between each other and $\Pr(X_i = 1) = \frac{1}{2}$.

We have $\mu = E[X] = E[\sum_{i=1}^{n} X_i] = n * \frac{1}{2} = \frac{n}{2}$. Now find δ with $(1+\delta)\mu = \frac{n}{2} + \frac{\sqrt{6nln(n)}}{2}$

So, we do the following:

$$(1+\delta)\mu = \frac{n}{2} + \frac{\sqrt{6nln(n)}}{2}$$

$$(1+\delta)\frac{n}{2} = \frac{n}{2} + \frac{\sqrt{6nln(n)}}{2}$$

$$(1+\delta)\frac{n}{2} * 2 = \frac{n + \sqrt{6nln(n)}}{2} * 2$$

$$(1+\delta)n = n + \sqrt{6nln(n)}$$

$$(\cancel{1} + \delta) = \cancel{1} + \frac{\sqrt{6nln(n)}}{n}$$

$$\delta = \frac{\sqrt{6nln(n)}}{n}$$

Now, we apply the bound as follows:

$$\Pr(X > (1+\delta)\mu) \leq e^{\frac{-\mu\delta^2}{3}}$$

$$= \Pr\left(X > \frac{n}{2} + \frac{\sqrt{6nln(n)}}{2}\right) \leq$$

$$\leq e^{\frac{-\frac{n}{2} * \left(\frac{\sqrt{6nln(n)}}{n}\right)^2}{3}}$$

$$\leq e^{\frac{-\frac{\cancel{n}}{\cancel{2}} * \left(\frac{6^{\cancel{2}}\cancel{n}ln(n)}{\cancel{n^2}}\right)}{\cancel{3}}}$$

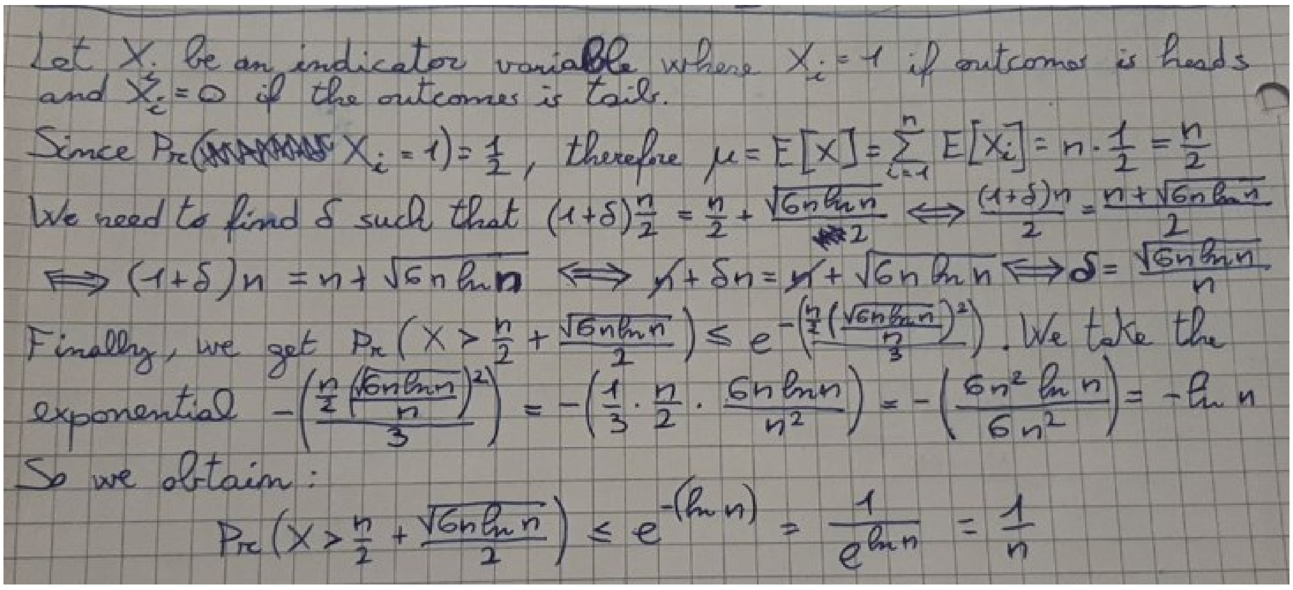
$$\leq e^{-\ln(n)}$$

$$\leq \frac{1}{n}$$

as the exercise wanted.

Written by Gabriel R.

For a handwritten solution (slightly different, more compact



Let X_i be an indicator variable where $X_i = 1$ if outcomes is heads and $X_i = 0$ if the outcomes is tails.

Since $\Pr(X_i = 1) = \frac{1}{2}$, therefore $\mu = E[X] = \sum_{i=1}^{n} E[X_i] = n \cdot \frac{1}{2} = \frac{n}{2}$

We need to find δ such that $(1+\delta)\frac{n}{2} = \frac{n}{2} + \frac{\sqrt{6n\ln n}}{2} \iff \frac{(1+\delta)n}{2} = \frac{n+\sqrt{6n\ln n}}{2}$

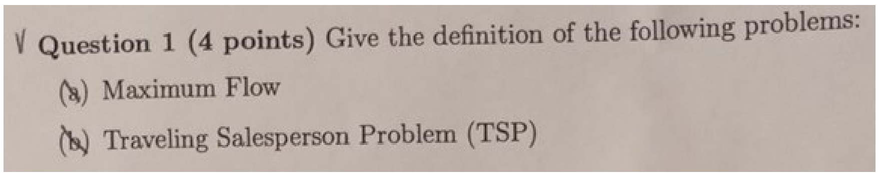
$\iff (1+\delta)n = n + \sqrt{6n\ln n} \iff \cancel{n} + \delta n = \cancel{n} + \sqrt{6n\ln n} \iff \delta = \frac{\sqrt{6n\ln n}}{n}$

Finally, we get $\Pr\left(X > \frac{n}{2} + \frac{\sqrt{6n\ln n}}{2}\right) \le e^{-\left(\frac{\frac{n}{2}\left(\frac{\sqrt{6n\ln n}}{n}\right)^2}{3}\right)}$. We take the exponential $-\left(\frac{\frac{n}{2}\left(\frac{\sqrt{6n\ln n}}{n}\right)^2}{3}\right) = -\left(\frac{1}{3} \cdot \frac{n}{2} \cdot \frac{6n\ln n}{n^2}\right) = -\left(\frac{6n^2\ln n}{6n^2}\right) = -\ln n$

So we obtain:

$$\Pr\left(X > \frac{n}{2} + \frac{\sqrt{6n\ln n}}{2}\right) \le e^{-(\ln n)} = \frac{1}{e^{\ln n}} = \frac{1}{n}$$

14.5 Exam of 24-06-2024


√ **Question 1 (4 points)** Give the definition of the following problems:

(a) Maximum Flow

(b) Traveling Salesperson Problem (TSP)

a) The maximum flow problem is a network optimization problem which, given a directed graph where each edge has a capacity, wants to find all possible paths from a single source $s \in V$ to a single sink $t \in V$, wants to find a flow of maximum value (the previous is said to be flow network) – maximum flow – while respecting capacity constraints of each edge.

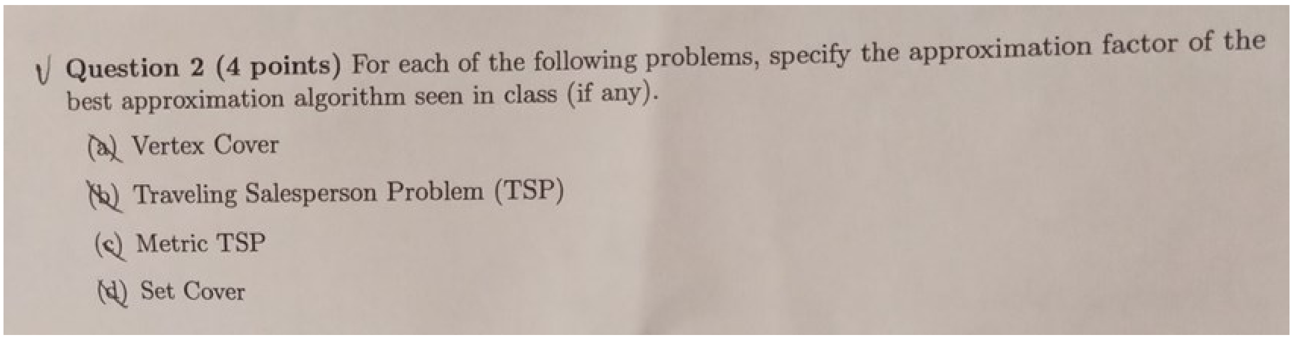
The flow is a function which considers capacity, conservation of flows, so to not surpass capacity of edge and capacity going in = capacity going out (flow conservation). The value of a flow is the sum of all flows going in and out vertices thanks to edges.

Consider the value of a flow is $|f| = \sum_{v \in V \ s.t. (s,v) \in E} f(s,v)$

The algorithm takes an augmenting path and changes the path flow, continuing until there is no other path from the source to the sink.

b) TSP is a classic optimization problem which asks, "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?". It's NP-Hard, since no poly-time algorithm exists to solve it and NP-Complete.

Its statement is the following: Given a complete undirected graph and a function $w: E \to \mathbb{R}^+$, output a cycle that passes through every vertex once ($T \subseteq E$), minimizing the cost of the tour $\sum_{e \in T} w(e)$


√ **Question 2 (4 points)** For each of the following problems, specify the approximation factor of the best approximation algorithm seen in class (if any).

(a) Vertex Cover

(b) Traveling Salesperson Problem (TSP)

(c) Metric TSP

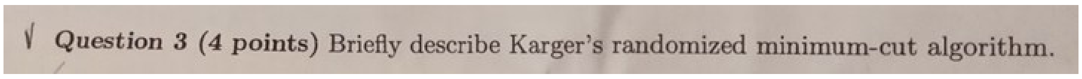
(d) Set Cover

a) 2

b) There is no approx-algo (NO)

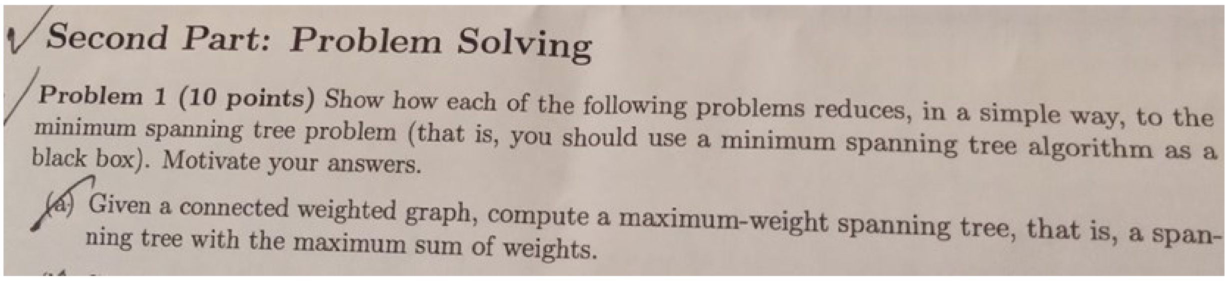
c) $\frac{3}{2}$

d) $\log(n)$


√ **Question 3 (4 points)** Briefly describe Karger's randomized minimum-cut algorithm.

Karger is a Monte Carlo (randomized algorithm which may fail) algorithm which considers and solves the problem of the minimum cut (cut of minimum size so to remove the minimum number of edges to make the graph disconnected). It revolves around the concept of "contraction" (have two vertices merge into each other), so choosing an edge at random, contract the two vertices of that edge and removing all of the edges incident to both vertices, keeping the smallest cut found.

The algorithm is probabilistic, meaning it may not always find the true minimum cut, but repeating it increases the probability of finding the correct result. The number of repetitions needed for a high probability of success is typically polynomial in the number of vertices.


✓ **Second Part: Problem Solving**

Problem 1 (10 points) Show how each of the following problems reduces, in a simple way, to the minimum spanning tree problem (that is, you should use a minimum spanning tree algorithm as a black box). Motivate your answers.

(a) Given a connected weighted graph, compute a maximum-weight spanning tree, that is, a spanning tree with the maximum sum of weights.

To reduce the problem of finding a maximum-weight spanning tree to a minimum spanning tree problem, we can use a simple weight transformation. Here's how we can approach this:

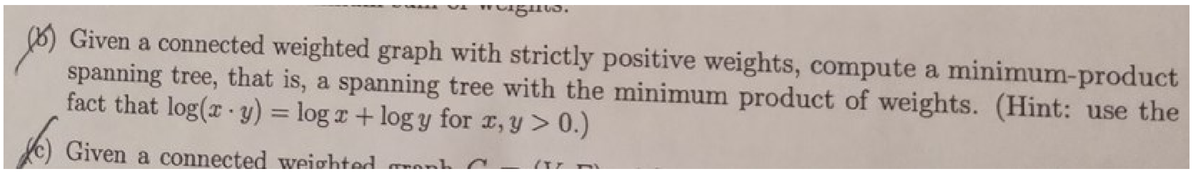
1. Take the original graph G and create a new graph G' with the same structure but transform each edge weight w to $-w$ (or to $W - w$, where W is a value larger than the maximum weight in the original graph).
2. Use any minimum spanning tree algorithm (like Kruskal's or Prim's) on G' as a "black box".
3. The minimum spanning tree of G' will correspond to the maximum spanning tree of G.

Motivation:

- By negating the weights (or subtracting from a large value), we invert the problem. The "cheapest" edges in G' correspond to the most expensive edges in G.
- Minimum spanning tree algorithms choose the cheapest edges that connect all vertices without forming cycles.
- Therefore, when applied to G', the algorithm will select the edges that were originally the heaviest in G, resulting in a maximum-weight spanning tree for the original problem.

So, in short (coming from the group):

- Find a MaxST → just multiply $x - 1$ every weight and then give it to MST, the result is a MaxST


(b) Given a connected weighted graph with strictly positive weights, compute a minimum-product spanning tree, that is, a spanning tree with the minimum product of weights. (Hint: use the fact that $\log(x \cdot y) = \log x + \log y$ for $x, y > 0$.)

(c) Given a connected weighted graph $G = (V, E)$

To reduce this to a minimum spanning tree problem:

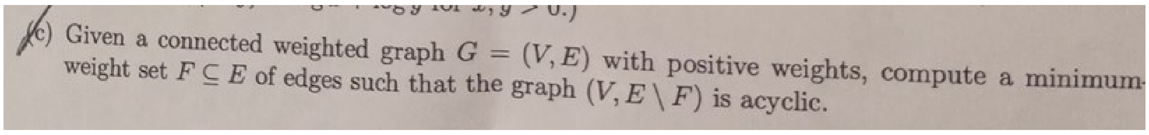
1. For each edge weight w, compute $log(w)$.
2. Use a standard minimum spanning tree algorithm on the graph with these transformed weights.

3. The resulting tree will be the minimum-product spanning tree for the original graph.

This works since thee logarithm transformation converts the product of weights to a sum of logarithms. Minimizing the sum of logarithms is equivalent to minimizing the product of the original weights. Such is motivated by the fact the logarithm is a monotonically increasing function for positive numbers.

So, in short (coming from the group):

- Find a $MinMultiplicationST$ → From the hint there was the solution, since $log(xy) = log(x) + log(y)$, just trasnform every weight into its log counterpart and then give it to MST, the result is a MinMultiplicationST



$\log y$ for $x, y > 0$.)

c) Given a connected weighted graph $G = (V, E)$ with positive weights, compute a minimum-weight set $F \subseteq E$ of edges such that the graph $(V, E \setminus F)$ is acyclic.

To reduce this to a minimum spanning tree problem:

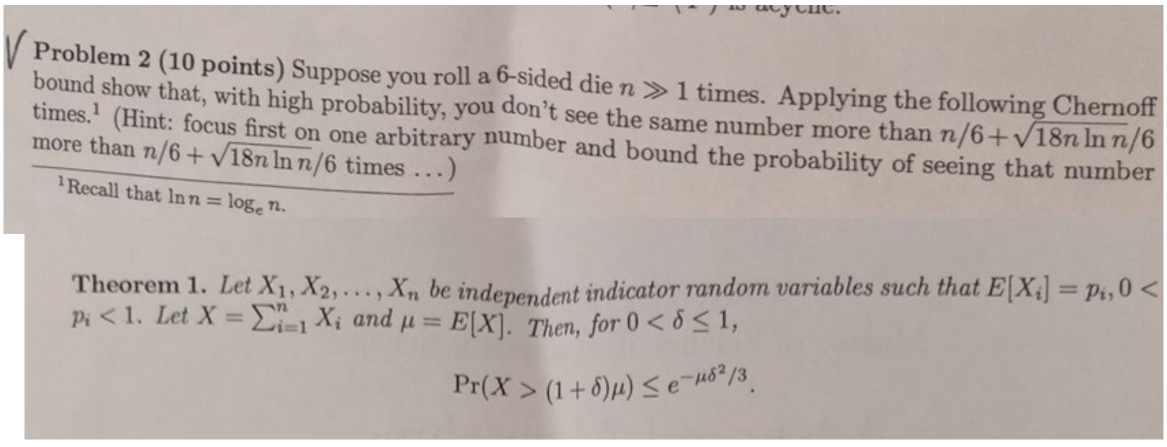
1. Instead of finding edges to remove to make the graph acyclic, we can find edges to keep that form a spanning tree.
2. Use a standard minimum spanning tree algorithm on the original graph G.
3. The set F will be all edges in E that are not in the minimum spanning tree.

This works since removing all edges not in the MST will leave an acyclic graph (the MST itself) – given a tree has no cycles to begin with as constructive property

The removed edges (F) will have the maximum weight among all sets that could be removed to leave a tree, which is equivalent to having the minimum weight among all sets whose removal leaves a tree.

So, in short (coming from the group):

- Find a subset of edges of min weigh s.t. there is no cycle → we can run point a) (MaxST) and then subtract MaxST from our graph, the result is a subset of edges of min weigh s.t. there is no cycle



$\setminus F)$ is acyclic.

Problem 2 (10 points) Suppose you roll a 6-sided die $n \gg 1$ times. Applying the following Chernoff bound show that, with high probability, you don't see the same number more than $n/6 + \sqrt{18n \ln n}/6$ times.[1] (Hint: focus first on one arbitrary number and bound the probability of seeing that number more than $n/6 + \sqrt{18n \ln n}/6$ times ...)

[1] Recall that $\ln n = \log_e n$.

Theorem 1. *Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that $E[X_i] = p_i, 0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then, for $0 < \delta \leq 1$,*

$$\Pr(X > (1+\delta)\mu) \leq e^{-\mu\delta^2/3}.$$

$X = \{1,2,3,4,5,6\}$ = dice's universe

Characterize the event $X_i = 1$ if we get a number, 0 otherwise

$\Pr(X_i = 1) = \frac{1}{6}$

$$X = \sum_{i=1}^{n} X_i = \sum_{i=1}^{n} E[X_i] = \mu = n * \frac{1}{6} = \frac{n}{6}$$

We find now δ:

$$(1+\delta)\mu = \frac{n}{6} + \frac{\sqrt{18nln(n)}}{6}$$

$$(1+\delta)\frac{n}{6} = \frac{n}{6} + \frac{\sqrt{18nln(n)}}{6}$$

$$(1+\delta)n = n + \sqrt{18nln(n)}$$

$$n + \delta n = n + \sqrt{18nln(n)}$$

$$\delta n = \sqrt{18nln(n)}$$

$$\delta = \frac{\sqrt{18nln(n)}}{n}$$

Apply the Chernoff Bound now.

$$\Pr\left(X > \frac{n}{6} + \frac{\sqrt{18nln(n)}}{6}\right) \le e^{\frac{-\mu\delta^2}{3}}$$

$$\le e^{\frac{-\frac{n}{6}\left(\frac{\sqrt{18nln(n)}}{n}\right)^2}{3}}$$

$$\le e^{\frac{-\frac{n}{6} * \frac{18nln(n)}{n^2}}{3}}$$

$$\le e^{\left(-\frac{n}{6}\right) * \frac{18nln(n)}{n^2} * \frac{1}{3}} = e^{-\ln(n)} = \frac{1}{e^{\ln(n)}} = \frac{1}{n}$$

The probability of seeing the same face is $\frac{1}{n}$.

It has to be applied with the Union Bound, which is for the "n" – in this case is 6. From the correction of the exam, we know this has to be applied for 6, so it is $6 * \frac{1}{n} = \frac{6}{n}$

The probability of NOT seeing the same face for all the dices for more than $\frac{n}{6} + \frac{\sqrt{18nln(n)}}{6}$ times is $1 - \frac{6}{n}$ (from the correction – see the group – this is the final result, on which to apply the Union Bound).

14.6 Exam of 22-06-2023

First Part: Theory Questions

Question 1 (4 points) Consider the following directed, weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding edge, and where the symbol '–' indicates the absence of the edge between the corresponding vertices.

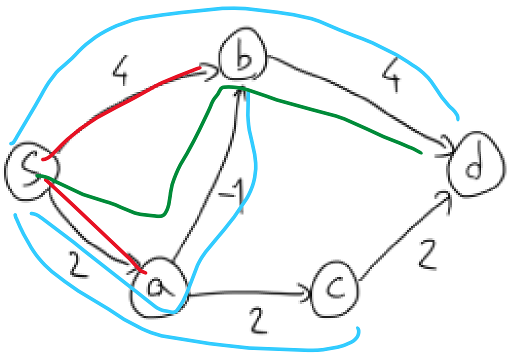
	s	a	b	c	d
s	-	2	4	-	-
a	-	-	-1	2	-
b	-	-	-	-	4
c	-	-	-	-	2
d	-	-	-	-	-

(a) Draw the graph.

(b) Run the Bellman-Ford algorithm on this graph, using vertex s as the source. You are to return the trace of the execution, i.e. a table with rows indexed by vertices and columns indexed by iteration indexes (starting from 0) where each entry contains the estimated distance between s and that vertex at that iteration.

Solution:

(a)



(b)

	0	1	2	3	4
s	0	0	0	0	0
a	∞	2	2	2	2
b	∞	4	1	1	1
c	∞	∞	4	4	4
d	∞	∞	8	5	5

Question 3 (4 points) Define the set cover problem and briefly describe the $O(\log n)$-approximation algorithm seen in class.

Solution: See the lecture notes.

Set cover is an optimization problem that models many problems requiring resources to be allocated. It aims to find the least number of subsets that cover some universal set.

Its inputs are:

- $I = (X, F)$ = instance of the set covering problem
- $X =$ set of elements of any kind, called "universe"
- $F \subseteq \{S: S \subseteq X\} = B(X)$
    a. B stands for "Boolean": set of all subsets of X

There is a constraint that needs to be always respected: $\forall x \in X, \exists\, S \in F: x \in S$ i.e., "F covers X"

Optimization problem: (smallest subset of F having its members covering all X) → find $F' \subseteq F$ s.t.

5) F' covers X
6) $\min\ |F'|$

Example:

$X = \{1,2,3,4,5\}$

$F = \{\{1,2,3\}, \{2,4\}, \{3,4\}, \{4,5\}\}$

$\Rightarrow F^* = \{\{1,2,3\}, \{4,5\}\}$

The greedy method works by picking at each stage, the set S that covers the greatest number of remaining elements that are uncovered:

- choose the subset that contains the largest number of uncovered elements
- remove from X those covered elements
- repeat until $X = \emptyset$

```
$Approx\_Set\_Cover(X, F)$
    $U = X$
    $F' = \emptyset$ // solution
    while $U \neq \emptyset$: do
        // take the set of F covering as many elements as possible
        let $S \in F = |S \cap U| = \max_{S' \in F}\{|S' \cap U|\}$
        $U \leftarrow U \setminus S$      // update the list of available elements, removing those from S
        $F \leftarrow F \setminus \{S\}$    // remove the sets already considered inside of F
        $F' \leftarrow F' \cup \{S\}$
    return $F'$
```

Second Part: Problem Solving

Problem 1 (9 points) Consider Dijkstra's algorithm seen in class, which returns the lengths of the shortest paths from a source vertex to all other vertices in directed graphs with nonnegative weights:

(a) Explain how to modify Dijkstra's algorithm to return the shortest paths themselves (and not just their lengths).

(b) Consider the following algorithm for finding shortest paths in a directed graph where edges may have negative weights: add the same large constant to each edge weight so that all the weights become nonnegative, then run Dijkstra's algorithm and return the shortest paths. Is this a valid method? Either prove that it works (i.e., the returned shortest paths are shortest paths in the original graph), or give a counterexample.

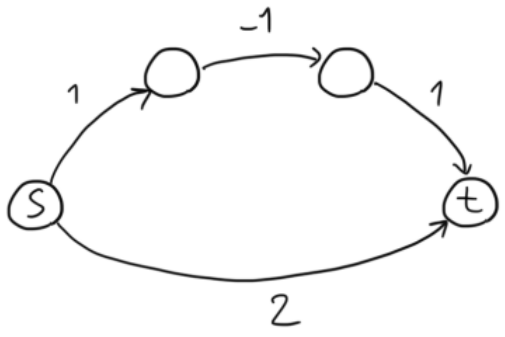
(c) Now let's switch to minimum spanning trees, and do the same: add the same large constant to each edge weight and then run Prim's algorithm. Either prove that the returned solution is a minimum spanning tree of the original graph, or give a counterexample.

Solution:

(a) Associate a pointer *predecessor*(v) to each vertex $v \in V$. When an edge $(v^\star, w^\star)$ is selected in an iteration of the main while loop, assign *predecessor*$(w^\star)$ to $v^\star$. Then, at the end of the algorithm, to reconstruct a shortest path from s to a vertex v, follow the *predecessor* pointers backward from v to s.

(b) Alas, it does not work. In other words, you cannot reduce the SSSP problem with general edge weights to the special case of nonnegative weights in this way. The problem is that different paths from one vertex to another might not have the same number of edges, hence if we add some number to each edge weight, the lengths of different paths can increase by different amounts, and therefore a shortest path in the new graph might be different than in the original

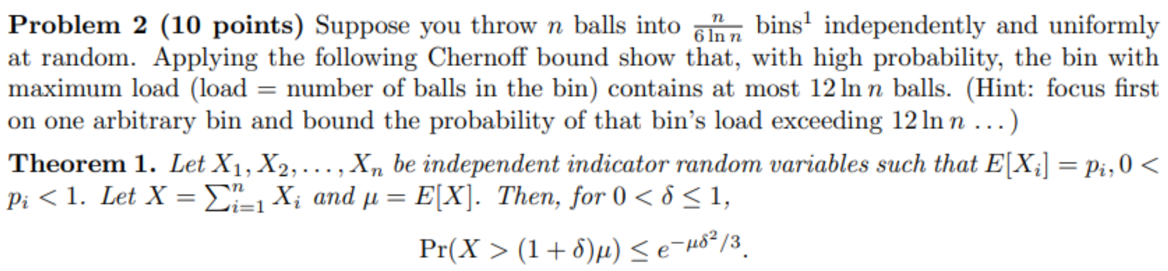
graph. For example, consider the following graph:



The shortest path from s to t is the 3-hop path on top; however, if we add 1 to every edge's weight, the shortest path from s to t changes, and becomes the direct s-t edge—which has length 3, while the 3-hop path on top has length 4. (We would arrive at the same conclusion no matter which value we choose to force the graph to have nonnegative edge weights.) Moreover, if the original graph has negative cycles, these would not be detected.

(c) For minimum spanning trees this works. In fact, all spanning trees have the same number of edges, hence if we add some number to each edge weight the total weight of different trees increases by the same amount.

ì



Problem 2 (10 points) Suppose you throw n balls into $\frac{n}{6\ln n}$ bins[1] independently and uniformly at random. Applying the following Chernoff bound show that, with high probability, the bin with maximum load (load = number of balls in the bin) contains at most $12\ln n$ balls. (Hint: focus first on one arbitrary bin and bound the probability of that bin's load exceeding $12\ln n$ ...)

Theorem 1. *Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that $E[X_i] = p_i, 0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then, for $0 < \delta \leq 1$,*

$$\Pr(X > (1+\delta)\mu) \leq e^{-\mu\delta^2/3}.$$

Consider $X_i = 1, 2, \ldots n$ since we can assign the i^{th} ball to any possible bin. So, this is uniform at random. The load of the specific bin is given by $X = \sum_{i=1}^{n} X_i$.

First, we have to find μ:

$$\mu = E[X] = \sum_{i=1}^{n} E[X_i] = n * \frac{6\ln(n)}{n} = 6\ln(n)$$

Since each ball is assigned to a bin chosen uniformly at random, we have

$$\Pr(X_i = 1) = \frac{1}{m} * n = \frac{1}{\frac{n}{6\ln(n)}} * n = \frac{6\ln(n)}{n}$$

To apply the Chernoff bound, we set $12\ln(n)$ equal to $(1+\delta)$ so:

$$(1+\delta)\mu = 12\ln(n)$$

$$(1+\delta)6\ln(n) = 12\ln(n)$$

$$(1+\delta) = 2$$

$$\delta = 1$$

Now, we apply the bound:

$$\Pr(X > (1+\delta)\mu) \leq e^{\frac{-\mu\delta^2}{3}}$$

$$\Pr(X > 1+1)\,6\ln(n)) \leq e^{-\frac{6\ln(n)}{3}}$$

$$\leq e^{-2\ln(n)}$$

Recall the property of exponentials and logarithms there, so:

$$\leq e^{\ln(n^{-2})}$$

Recall from the exercise that $\ln(n) = \log_e(n)$

So, we have:

$$e^{\ln(n^{-2})} = \frac{1}{n^2}$$

as the exercise wanted. We showed with high probability the bin with maximum load containing at most $12\ln(n)$ balls. We applied this for *one* bin, so we have to use now the union bound; simply use the previous result multiplying by all bins, so $m = \frac{n}{6n(\ln(n))}; \frac{n}{6\ln(n)} * \frac{1}{n^2} = \frac{1}{6n\ln(n)}$

To characterize the *no bin will exceed*, use the complement event → $1 - \frac{1}{6n\ln(n)} = 1 - o\left(\frac{1}{n}\right)$

Written by Gabriel R.

14.7 EXAM OF 06-07-2023

Question 1 (4 points) Give a list of graph problems that can be solved in time $O(m+n)$ applying DFS or BFS. Your list should include at least four problems to receive full points.

DFS/BFS can be used to solve the following problems:

- Test if graph is connected
- Find connected components
- Find a $s-t$ path
- Find a cycle, if it exists
- Find a spanning tree, if graph is connected

Complexity for both: $O(n+m)$

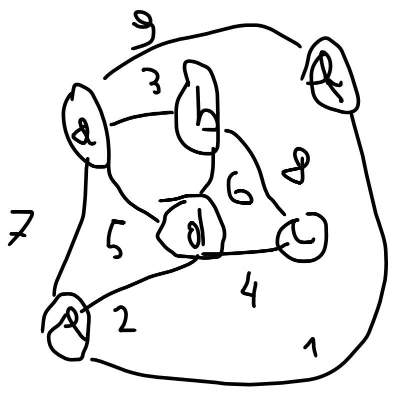
Question 2 (4 points) Consider the following weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding edge, and where the symbol '$-$' indicates the absence of the edge between the corresponding vertices.

	a	b	c	d	e	f
a	-	3	-	5	7	9
b		-	8	6	-	-
c			-	4	-	-
d				-	2	-
e					-	1
f						-

(a) Draw the graph.

(b) List the edges of the minimum spanning tree in the order they are selected by Kruskal's algorithm.

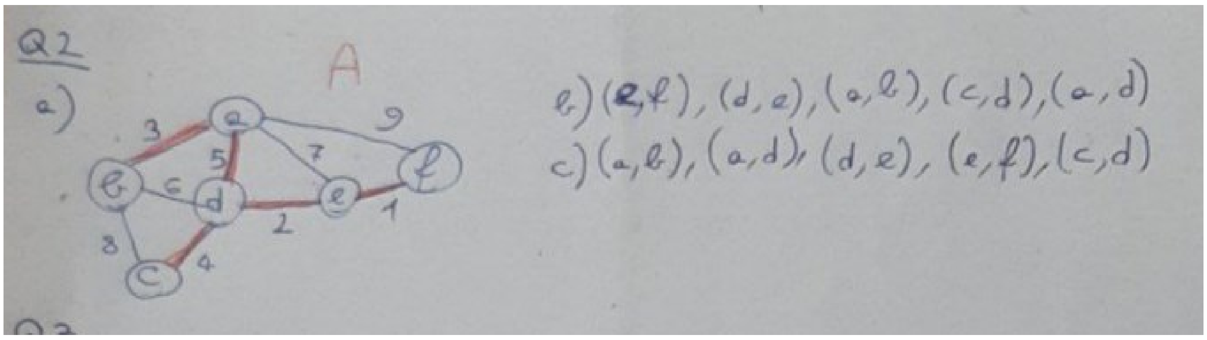
(c) List the edges of the minimum spanning tree in the order they are selected by Prim's algorithm starting at vertex a.



b) Kruskal: sort by weight in ascending order, then select edges: $\{e,f\},\{e,d\},\{a,b\},\{d,c\},\{a,d\}$

c) Prim: start from a source vertex then grow a spanning tree from there: $\{a,b\},\{a,d\},\{d,e\},\{e,f\},\{c,d\}$

Alternatively, a better graph here:


Q2
a)
A
3
9
5
7
6
8
2
1
4
a
b
c
d
e
f
b) (e,f), (d,e), (a,b), (c,d), (a,d)
c) (a,b), (a,d), (d,e), (e,f), (c,d)

Question 3 (4 points) For each of the following problems, say whether it is NP-hard or not and, if not, specify the complexity of the best algorithm seen in class.

(a) Maximum flow

(b) All-pairs shortest paths

(c) Vertex cover

(d) Single-source shortest paths

(a) $O(m|f^*|)$

(b) $O(n^3)$

(c) NP-hard

(d) $O(n * m)$

Second Part: Problem Solving

Exercise 1 (10 points) In the *bin packing* problem we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. This problem is NP-hard; this exercise asks you to analyze the following simple approximation algorithm, where the input is an array $W[1 \dots n]$ of weights and the output is the number of bins used.

```
ApproxBinPacking(W[1...n])
   b = 1               \\ b = current bin
   Total[1] = W[1]   \\ Total[b] = total weight of bin b
   for i = 2 to n do
      if Total[b] + W[i] <= 1 then    \\ item i fits into the current bin
         Total[b] = Total[b] + W[i]   \\ item i is placed inside it
      else
         b = b + 1          \\ a new bin (initially empty) is opened
         Total[b] = W[i]   \\ item i is placed inside this new bin
   return b
```

(a) Prove that this algorithm is a 2-approximation algorithm. (Hint: consider pairs of bins...)

(b) Prove a lower bound to the approximation factor of this algorithm. To receive full points the lower bound should be 2.

(a) (The first part gives you the code; to resemble what the professor did in his examples of exams exercises, we write the following). First of all, we observe that, since $Total \leq 1$ and a value $W[i]$ is added to $Total$ only if $Total[b] + W[i] \leq 1$, then the returned value is always the cost of a feasible solution. If S^* denotes the optimal solution, we need to prove $\frac{S^*}{Total} \leq 2 \rightarrow \frac{S^*}{S'} \leq 2$.

- Case 1: The algorithm returns out of the for loop; hence $Total = \sum_{i=1}^{n} W[i] \leq 1$, that is $Total = S^*$ and thus $\frac{S^*}{Total} = 1 \leq 2$

- Case 2: The algorithm returns from inside the for loop: hence there exists and index i' such that $Total + W[i'] > 1$. Observe that:

$$W[i'] < W[i] < Total$$

and hence $2 * Total > Total + W[i'] > 1$ that is $Total > \frac{1}{2} \geq \frac{S^*}{2}$.

To give a more concrete explanation (done formally):

The algorithm starts with one bin and proceeds through each item. If the current item fits in the existing bin (total weight including the item remains ≤ 1), it is added to the current bin. Otherwise, a new bin is opened for this item.

The structure to prove would be the following, since this is a *minimization* problem:

$$|S'| \leq 2|S^*|$$

or completely:

$$|S'| \leq \cdots \leq 2|S^*| \rightarrow \frac{|S'|}{|S^*|} \leq 2$$

The algorithm places the next item in the current bin if it fits; otherwise, it opens a new bin. Thus, each bin (except possibly the last one) has a total weight of more than 0.5.

This is because if the next item could not fit, the current bin's weight is more than $1 - weight\ of\ next\ item$ and since each item's weight is at most 1. Each bin is at least half full, except for the last one.

In the optimal packing, each bin is packed as much as possible without exceeding the total weight of 1. The optimal number of bins would select all bins to ensure the sum is actually 1.

The greedy choice would consider all bins, so to select everything if possible → $k > \frac{1}{2}$

So, we have → $\frac{k-1}{2} < \frac{k}{2} < \sum$ weights.

Specifically, considering the total weight W of all items, the optimal solution, at its absolute densest, would pack bins with a total weight of 1 per bin. Therefore, $|S^*| \geq |W^*|$.

To ensure this, we would select pairs of bins, while the greed would simply select all bins. Since each bin in the algorithm has a total weight exceeding 0.5 (except possibly the last one), the number of bins used by the algorithm $|S'|$ is at most $2W$. Hence, $|S'| \leq 2|W|$. Given $|S^*| \geq |W|$, we have $|S'| \leq 2|S^*|$

Let's structure completely the proof:

- Lower bound to the cost of S^*

As said, the optimal solution would be bounded at least by the number of bins, given each one is chosen, with weight $> 0.5\ \left(\frac{1}{2}\right)$. If we pair the bins, it's clear we should consider the greedy choice would be at least half → $|S'| \leq 2|S^*|$

- Upper bound to the cost of $|S'|$

By construction, the optimal solution S^* packs items such that the total weight in each bin is maximally utilized but does not exceed 1. This means the greedy solution is exactly half of the full pairs of bins, so:

$$|S'| \leq 2|S^*|$$

(b)

Consider the array:

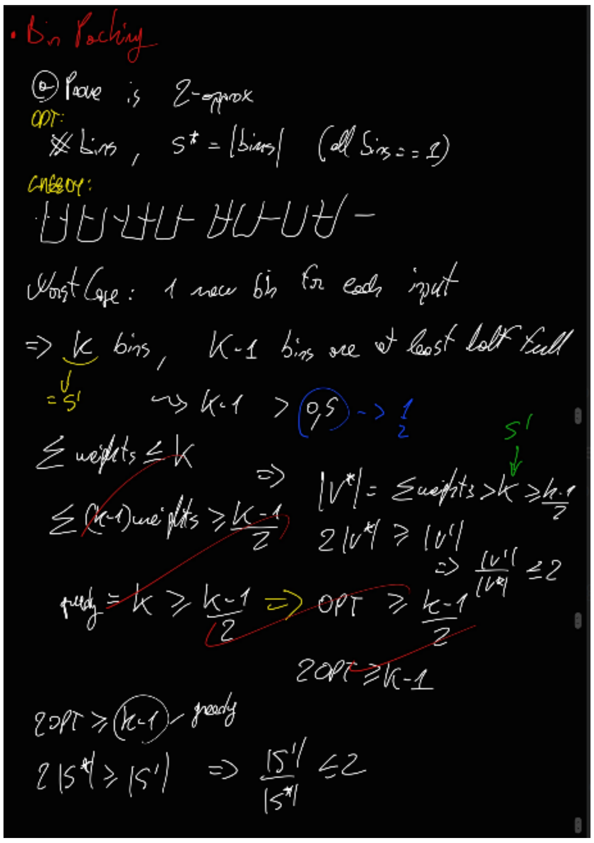
$$S[0, 1, 1, 0 \dots .]$$

$$|S'| \rightarrow take\ all\ bins$$

$$|S^*| \rightarrow take\ a\ couple\ of\ bins, where\ same\ weights\ are\ taken\ half\ for\ each\ couple \rightarrow \frac{n}{2}$$

$$\frac{|S'|}{|S^*|} = \frac{n}{\frac{n}{2}} = 2$$

In the group, the following solution was given:



• Bin Packing

(a) Prove is 2-approx

OPT:

# bins, $S^* = |bins|$ (all $S_i s == 1$)

GREEDY:

Worst Case: 1 new bin for each input

$\Rightarrow \underbrace{k}$ bins, $k-1$ bins are at least half full

$= S'$

$\rightsquigarrow k-1 > \boxed{0{,}5} \rightarrow \frac{1}{2}$

$\sum weights \leq k$ $\Rightarrow$ $|V^*| = \sum weights > \overset{S'}{k} \geq \frac{k-1}{2}$

$\sum (k-1) weights \geq \frac{k-1}{2}$ $\quad 2|V^*| \geq |V'|$

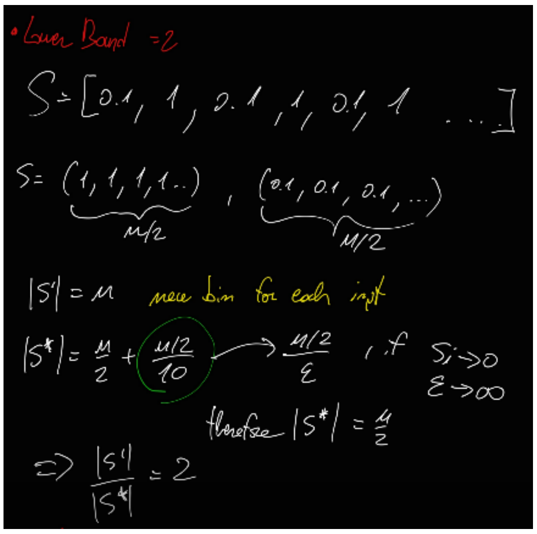
$\Rightarrow \frac{|V'|}{|V^*|} \leq 2$

$greedy = k \geq \frac{k-1}{2} \Rightarrow OPT \geq \frac{k-1}{2}$

$2OPT \geq k-1$

$2OPT \geq \boxed{k-1}$ – greedy

$2|S^*| \geq |S'| \Rightarrow \frac{|S'|}{|S^*|} \leq 2$



• Lower Band = 2

$S = [0.1, 1, 0.1, 1, 0.1, 1 \dots]$

$S = \underbrace{(1, 1, 1, 1 \dots)}_{n/2}, \underbrace{(0.1, 0.1, 0.1, \dots)}_{n/2}$

$|S'| = n$ new bin for each input

$|S^*| = \frac{n}{2} + \boxed{\frac{n/2}{10}} \rightarrow \frac{n/2}{\varepsilon}$, if $S_i \rightarrow 0$, $\varepsilon \rightarrow \infty$

therefore $|S^*| = \frac{n}{2}$

$\Rightarrow \frac{|S'|}{|S^*|} = 2$

Written by Gabriel R.

Exercise 2 (10 points) Suppose you have a randomized algorithm for a minimization problem A that returns the correct output with probability at least $1/n$, where n is the input size. Show how to obtain an algorithm for A that returns the correct output with high probability. (Hint: for the analysis use this inequality: $(1+x/y)^y \leq e^x$ for $y \geq 1$, $y \geq x$.)

Here, we use Karger since the hint uses that inequality and the only point in the program we saw that is exactly that analysis – so, that's why we use that in place of a "normal" Chernoff Bound.

Characterize the event of getting a probability *at least* $\frac{1}{n}$ using Karger's analysis; run different times the analysis and fix a constant d, $d > 0$:

$$\Pr\left(X > \frac{1}{n}\right) > \frac{1}{n^d}$$

Since the probability is at least $\frac{1}{n}$, so we characterize using Karger. Here, we will characterize the probability of failure (the complement with respect to the previous, so $1 - \frac{1}{n}$, running k times to reduce the error probability.

We want to find a value for k such that $Pr\left(1 - \frac{1}{n}\right)^k \leq \frac{1}{n^d}$. In this case, it's standard the use of this inequality:

$$\left(1 + \frac{x}{y}\right)^y \leq e^x, y \geq 1, y \geq x$$

This inequality is derived from the exponential function and the binomial expansion. It represents an upper bound on the expression $\left(1 + \frac{x}{y}\right)^y$, showing that it grows slower than e^x.

Now, we use Karger's analysis, in place of using $k = n^2$ we use $k = n$ and everything comes naturally.

By choosing $k = dn\ln(n)$ it follows that (it holds $d = 1$ coming from Karger):

$$\left(1 - \frac{1}{n}\right)^{k=n} \leq e^{-1} = \frac{1}{e} \rightarrow \text{is } \underline{\text{not}} \text{ in the form } \frac{1}{n^d}$$

Recall the following from the Karger analysis (choice of k and rest of reasoning is that):

$$e^{-\ln(n^d)} = \frac{1}{n^d}$$

Continuing using Karger and the inequality:

$$\left(\left(1 - \frac{1}{n}\right)^n\right)^{\ln(n^d)} = \left(1 - \frac{1}{n}\right)^{n\ln(n)}$$

Let's wrap up:

$$\left(1 - \frac{1}{n}\right)^{k=n\ln(n)} = \left(\left(1 - \frac{1}{n}\right)^n\right)^{\ln(n)}$$

$$\leq (e^{-1})^{\ln(n)} = e^{-\ln(n)} = \frac{1}{n}$$

<u>For reference from theory, Karger's analysis is this.</u>

$\Pr\,(\textit{the k runs of FULL\_CONTRACTION do not return the min cut}) \le \left(1 - \frac{2}{n^2}\right)^k \le \frac{1}{n^d}$ for some constant $d > 0$

The previous one is the probability of an unsuccessful event, so we want it very low, something like $\frac{1}{n^d}$.

We want to find a value for k such that $\left(1 - \frac{2}{n^2}\right)^k \le \frac{1}{n^d}$. In this case, it's standard the use of this inequality:

$$\left(1 + \frac{x}{y}\right)^y \le e^x, y \ge 1, y \ge x$$

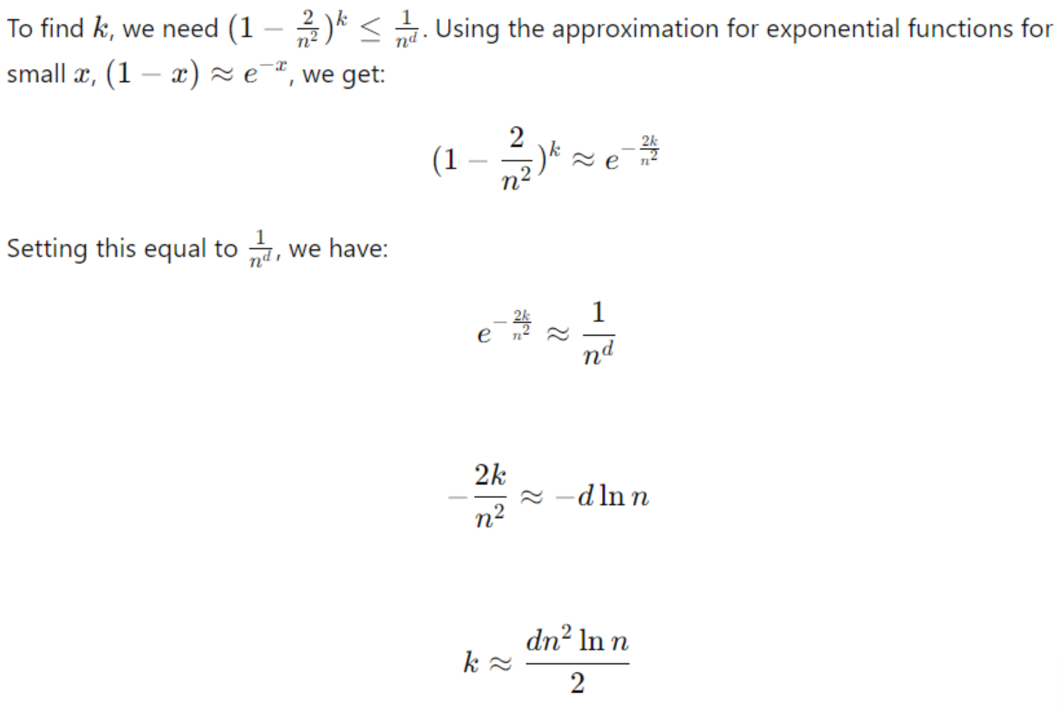
This inequality is derived from the exponential function and the binomial expansion. It represents an upper bound on the expression $\left(1 + \frac{x}{y}\right)^y$, showing that it grows slower than e^x. The probability of not contracting the minimum cut in each iteration needs to be bounded and manipulated to ensure the overall algorithm's success probability is high.

By choosing $k = \frac{dn^2 \ln(n)}{2}$ it follows that:

$$\left(\left(1 - \frac{2}{n^2}\right)^{n^2}\right)^{\ln(n^d)} \le e^{-\ln(n^d)} = \frac{1}{n^d}$$

Given I am curious, I asked myself: why exactly that value for k?

Consider the probability of success if $\frac{2}{n^2}$ while the failure is, by complement, $1 - \frac{2}{n^2}$ which, amplified by k runs, becomes $\left(1 - \frac{2}{n^2}\right)^k$. The constant d is the desired level of confidence to keep the wanted threshold (in this case $\frac{1}{n^d}$) as low as possible. Then, using some good old GPT-4:



To find k, we need $(1 - \frac{2}{n^2})^k \le \frac{1}{n^d}$. Using the approximation for exponential functions for small x, $(1 - x) \approx e^{-x}$, we get:

$$(1 - \frac{2}{n^2})^k \approx e^{-\frac{2k}{n^2}}$$

Setting this equal to $\frac{1}{n^d}$, we have:

$$e^{-\frac{2k}{n^2}} \approx \frac{1}{n^d}$$

$$-\frac{2k}{n^2} \approx -d \ln n$$

$$k \approx \frac{dn^2 \ln n}{2}$$

Moving on:

$$\left(1-\frac{2}{n^2}\right)^{k=n^2} \leq e^{-2} = \frac{1}{e^2} \rightarrow \text{is } \underline{\text{not}} \text{ in the form } \frac{1}{n^d}$$

Recall the following:

$$e^{-\ln(n^d)} = \frac{1}{n^d}$$

Let's apply that:

$$\left(\left(1-\frac{2}{n^2}\right)^{n^2}\right)^{\ln(n^d)} = \left(1-\frac{2}{n^2}\right)^{n^2 \ln(n^d)}$$

Let's wrap up (here, in the prof. notes, d magically disappears, but I assume it to be 1 so this works):

$$\left(1-\frac{2}{n^2}\right)^{\boxed{k=\frac{dn^2\ln(n^d)}{2}}} = \left(\left(1-\frac{2}{n^2}\right)^{n^2}\right)^{\frac{\ln(n^d)}{2}}$$

$$\leq (e^{-2})^{\frac{\ln(n^d)}{2}} = e^{-\ln(n^d)} = n^d = \frac{1}{n^d}$$

Then, by choosing that value for k the Karger's algorithm succeeds with high probability:

$$\Rightarrow \Pr(KARGER\ succeeds) > 1-\frac{1}{n^d}$$

So, in the end, the Karger algorithm accumulates the size of the min-cut with probability at least $\frac{1}{n^d}$.

14.8 Exam of 30-08-2023

Question 1 (4 points) Let $G = (V, E)$ be an undirected graph with n vertices and m edges. Given two vertices $s, t \in V$, briefly describe how to find, if it exists, a path from s to t in time $O(n + m)$.

Quoting the theory here (same exercise):

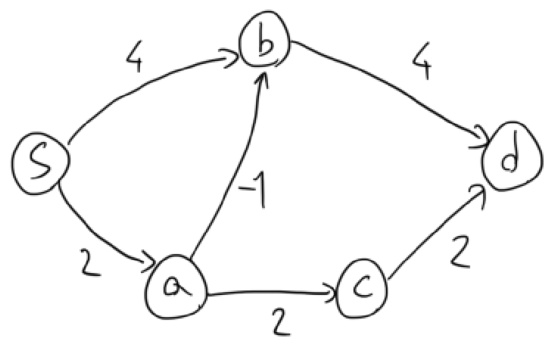
- $\forall v \in V$ add a field $L_V[v].parent$
- Modify $DFS(G, v)$ s.t. when a $DISCOVERY\ EDGE\ (v, w)$ is labeled
  - then $L_V[w].parent = v$ (v is parent of w in $DISCOVERY\ EDGE$ tree)
- Run $DFS(G, s)$. Check if t has been visited
  - NO: then return "No path"
  - YES: starting from t, follow the "parent" label, so as to build a path from t to s
- Complexity: $O(m_s)$ where m_s is the number of edges of s connected component

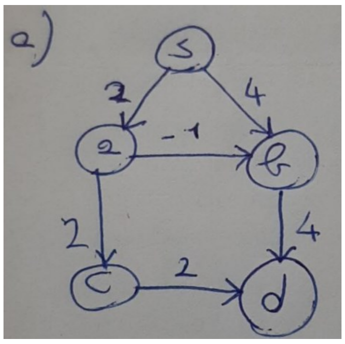
Question 2 (4 points) Consider the following directed, weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding edge, and where the symbol '–' indicates the absence of the edge between the corresponding vertices.

	s	a	b	c	d
s	-	2	4	-	-
a	-	-	-1	2	-
b	-	-	-	-	4
c	-	-	-	-	2
d	-	-	-	-	-

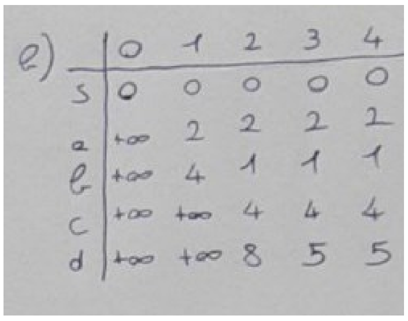
(a) Draw the graph.

(b) Run the Bellman-Ford algorithm on this graph, using vertex s as the source. You are to return the trace of the execution, i.e. a table with rows indexed by vertices and columns indexed by iteration indexes (starting from 0) where each entry contains the estimated distance between s and that vertex at that iteration.



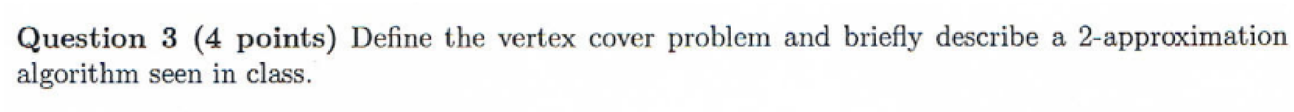


	0	1	2	3	4
s	0	0	0	0	0
a	∞	2	2	2	2
b	∞	4	1	1	1
c	∞	∞	4	4	4
d	∞	∞	8	5	5



b)

	0	1	2	3	4
s	0	0	0	0	0
a	$+\infty$	2	2	2	2
b	$+\infty$	4	1	1	1
c	$+\infty$	$+\infty$	4	4	4
d	$+\infty$	$+\infty$	8	5	5



Question 3 (4 points) Define the vertex cover problem and briefly describe a 2-approximation algorithm seen in class.

The vertex cover (also called completely "minimum vertex cover") is the a. minimum number of vertices that "touches" all edges. A 2-approximation algorithm designed in class was the one used for the TSP problem

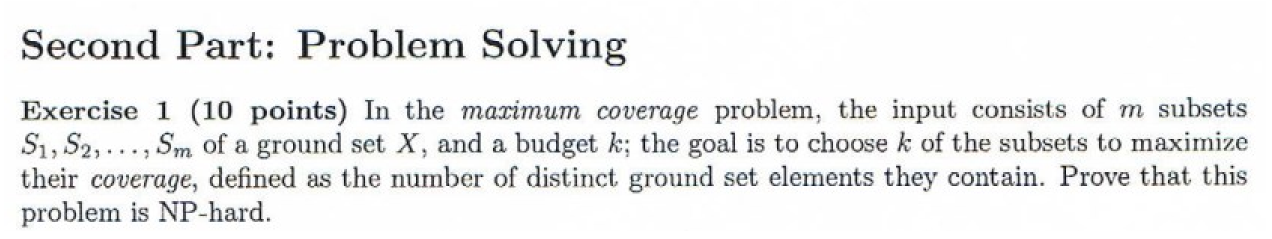
We use a greedy algorithm structured like the following:

- Choose *any* edge
- Add its endpoints to the solution
- "Remove" the covered edges
- Repeat

We'll show that this is a 2-approximation algorithm (which returns a solution whose cost is at most twice the optimal):

```
procedure $Approx\_Vertex\_Cover(G)$
    $V' = \emptyset$
    $E' = E$
    while $E' \neq \emptyset$: do
        $Let\ (u, v)\ be\ an\ arbitrary\ edge\ of\ E'$
        $V' = V' \cup \{u, v\}$
        $E' = E' \setminus \{(u, z), (v, w)\}$
        $//\ remove\ edges\ that\ have\ u\ and\ v\ as\ endpoints$
    return $V'$
```

Complexity: $O(n + m)$



Second Part: Problem Solving

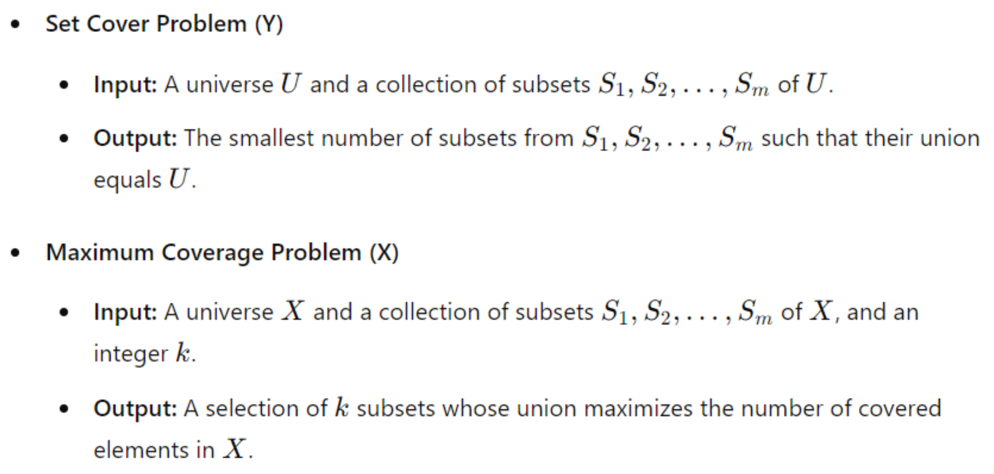
Exercise 1 (10 points) In the *maximum coverage* problem, the input consists of m subsets $S_1, S_2, \ldots, S_m$ of a ground set X, and a budget k; the goal is to choose k of the subsets to maximize their *coverage*, defined as the number of distinct ground set elements they contain. Prove that this problem is NP-hard.

The most similar problem is Set Cover; given a universe U and a collection of subsets $S_1, S_2, \ldots S_n$ of U, the goal is to find the smallest number of subsets such that every element in U is covered by at least one of the selected subsets. A reference for this specific reduction can be also found here.

To do the reduction:

- Take an instance of Set Cover and use the same universe U with subsets $S_1, S_2, \ldots S_n$ for the maximum coverage problem

Specifically:



- **Set Cover Problem (Y)**
  - **Input:** A universe U and a collection of subsets $S_1, S_2, \ldots, S_m$ of U.
  - **Output:** The smallest number of subsets from $S_1, S_2, \ldots, S_m$ such that their union equals U.
- **Maximum Coverage Problem (X)**
  - **Input:** A universe X and a collection of subsets $S_1, S_2, \ldots, S_m$ of X, and an integer k.
  - **Output:** A selection of k subsets whose union maximizes the number of covered elements in X.

The reduction process would work as follows:

- Use the universe U from Set Cover as universe X for Maximum Coverage ($X = U$)
- For any given k representing the number of subsets, use the same subsets $S_1, S_2, \ldots S_n$, this is the size of the optimal solution for Set Cover and Maximum Coverage alike
  - If Maximum Coverage can be solved in poly-time for any k, by trying all possible k from 1 to m, we can determine the smallest k such that the maximum coverage is $|U|$
- Evaluate Maximum Coverage for all possible values of k from 1 to m

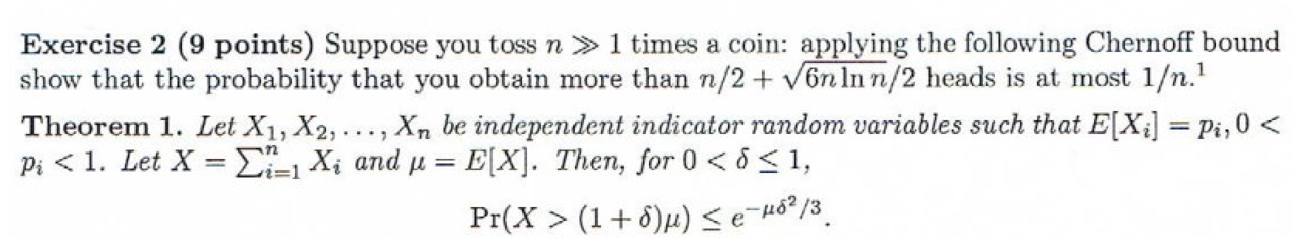
To prove this is correct:

- if Set Cover has a solution for some k, it means the k subsets cover the entire universe U and applying this to Maximum Coverage will yield full coverage of U for the selected k, showing the maximum coverage is $|U|$; infact, this ensures the optimal selection of the correct number of subsets so to achieve the correct solution with respect to the number of elements of Coverage, ensuring a good enough cover for size k

- if Maximum Coverage can be solved efficiently for any k, this means we can iterate through the subsets $k = 1$ to m (where m is the number of subsets). The smallest k that results in Maximum Coverage equaling $|U|$ corresponds to the smallest k solving Set Cover

    a. Since the problem itself is NP-Hard, the ability to solve it using MCP implies that MCP must also be NP-hard, since we can check and evaluate Maximum Coverage for each subset k up to m and this happens since $\forall k$ up to m we check if the coverage equals $|U|$ and take the smallest one

This proof clearly shows how a polynomial-time solution to Maximum Coverage would enable a polynomial-time solution to Set Cover, establishing the NP-hardness of Maximum Coverage through reduction defined precisely.


Exercise 2 (9 points) Suppose you toss $n \gg 1$ times a coin: applying the following Chernoff bound show that the probability that you obtain more than $n/2 + \sqrt{6n \ln n}/2$ heads is at most $1/n$.[1]

Theorem 1. *Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that $E[X_i] = p_i, 0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then, for $0 < \delta \leq 1$,*

$$\Pr(X > (1+\delta)\mu) \leq e^{-\mu\delta^2/3}.$$

First and foremost, $X_1 = 1$ if coin is tail, 0 otherwise. $X_i's$ are independent between each other and $\Pr(X_i = 1) = \frac{1}{2}$.

We have $\mu = E[X] = E[\sum_{i=1}^{n} X_i] = n * \frac{1}{2} = \frac{n}{2}$. Now find δ with $(1+\delta)\mu = \frac{n}{2} + \frac{\sqrt{6nln(n)}}{2}$

So, we do the following:

$$(1+\delta)\mu = \frac{n}{2} + \frac{\sqrt{6nln(n)}}{2}$$

$$(1+\delta)\frac{n}{2} = \frac{n}{2} + \frac{\sqrt{6nln(n)}}{2}$$

$$(1+\delta)\frac{n}{2} * 2 = \frac{n + \sqrt{6nln(n)}}{2} * 2$$

$$(1+\delta)n = n + \sqrt{6nln(n)}$$

$$(\cancel{1} + \delta) = \cancel{1} + \frac{\sqrt{6nln(n)}}{n}$$

$$\delta = \frac{\sqrt{6nln(n)}}{n}$$

Now, we apply the bound as follows:

$$\Pr(X > (1+\delta)\mu) \leq e^{\frac{-\mu\delta^2}{3}}$$

$$= \Pr\left(X > \frac{n}{2} + \frac{\sqrt{6nln(n)}}{2}\right) \leq$$

$$\leq e^{\frac{-\frac{n}{2} * \left(\frac{\sqrt{6nln(n)}}{n}\right)^2}{3}}$$

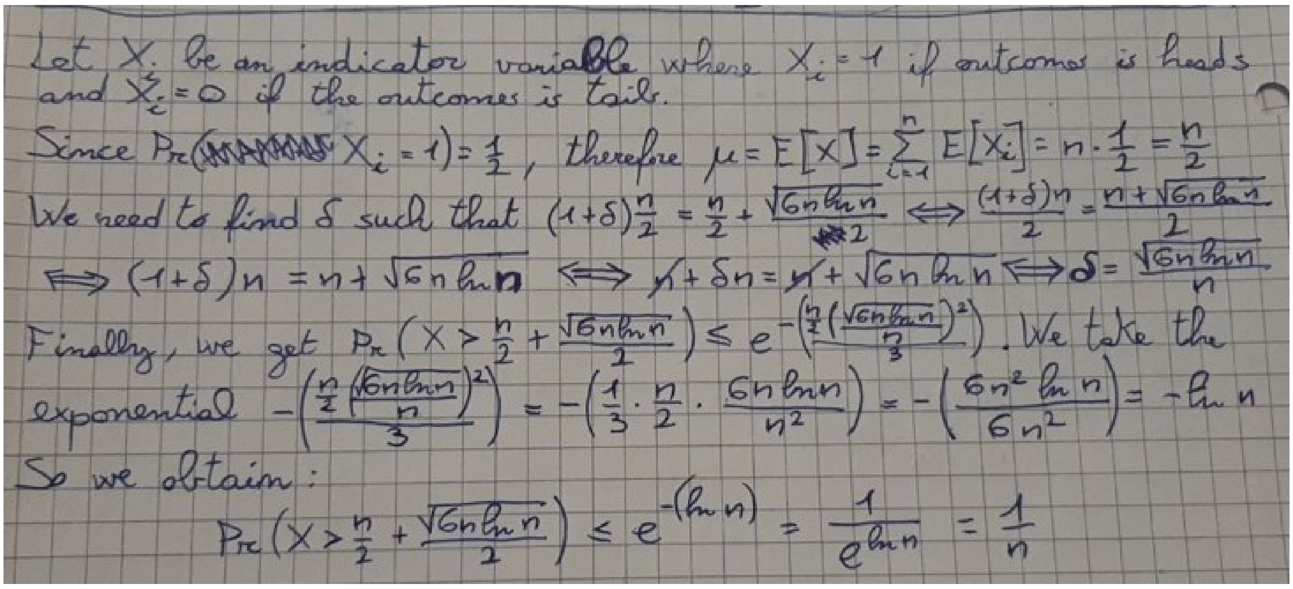
$$\leq e^{\frac{-\frac{\cancel{n}}{\cancel{2}} * \left(\frac{\cancel{6^2 n}ln(n)}{\cancel{n^2}}\right)}{\cancel{3}}}$$

$$\leq e^{-\ln(n)}$$

$$\leq \frac{1}{n}$$

as the exercise wanted.

For a handwritten solution (slightly different, more compact):



Let X_i be an indicator variable where $X_i = 1$ if outcomes is heads and $X_i = 0$ if the outcomes is tails.

Since $Pr(X_i = 1) = \frac{1}{2}$, therefore $\mu = E[X] = \sum_{i=1}^{n} E[X_i] = n \cdot \frac{1}{2} = \frac{n}{2}$

We need to find δ such that $(1+\delta)\frac{n}{2} = \frac{n}{2} + \frac{\sqrt{6n \ln n}}{2} \iff \frac{(1+\delta)n}{2} = \frac{n+\sqrt{6n\ln n}}{2}$

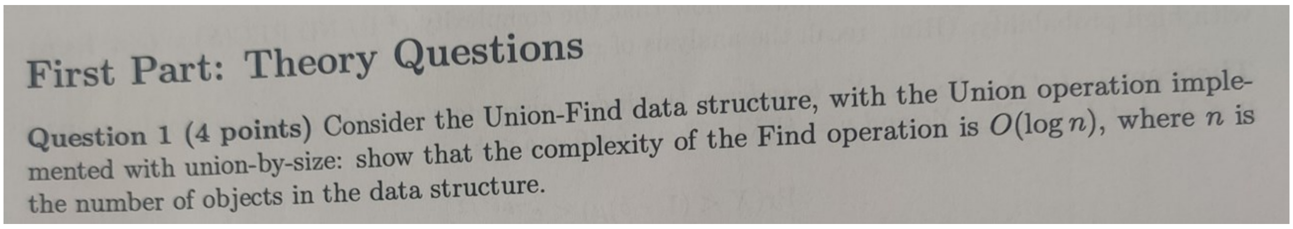
$\iff (1+\delta)n = n + \sqrt{6n \ln n} \iff \cancel{n} + \delta n = \cancel{n} + \sqrt{6n\ln n} \iff \delta = \frac{\sqrt{6n\ln n}}{n}$

Finally, we get $Pr\left(X > \frac{n}{2} + \frac{\sqrt{6n\ln n}}{2}\right) \leq e^{-\left(\frac{\frac{n}{2}\left(\frac{\sqrt{6n\ln n}}{n}\right)^2}{3}\right)}$. We take the exponential $-\left(\frac{\frac{n}{2}\left(\frac{\sqrt{6n\ln n}}{n}\right)^2}{3}\right) = -\left(\frac{1}{3} \cdot \frac{n}{2} \cdot \frac{6n\ln n}{n^2}\right) = -\left(\frac{6n^2 \ln n}{6n^2}\right) = -\ln n$

So we obtain:

$$Pr\left(X > \frac{n}{2} + \frac{\sqrt{6n\ln n}}{2}\right) \leq e^{-(\ln n)} = \frac{1}{e^{\ln n}} = \frac{1}{n}$$

14.9 EXAM OF 13-09-2023

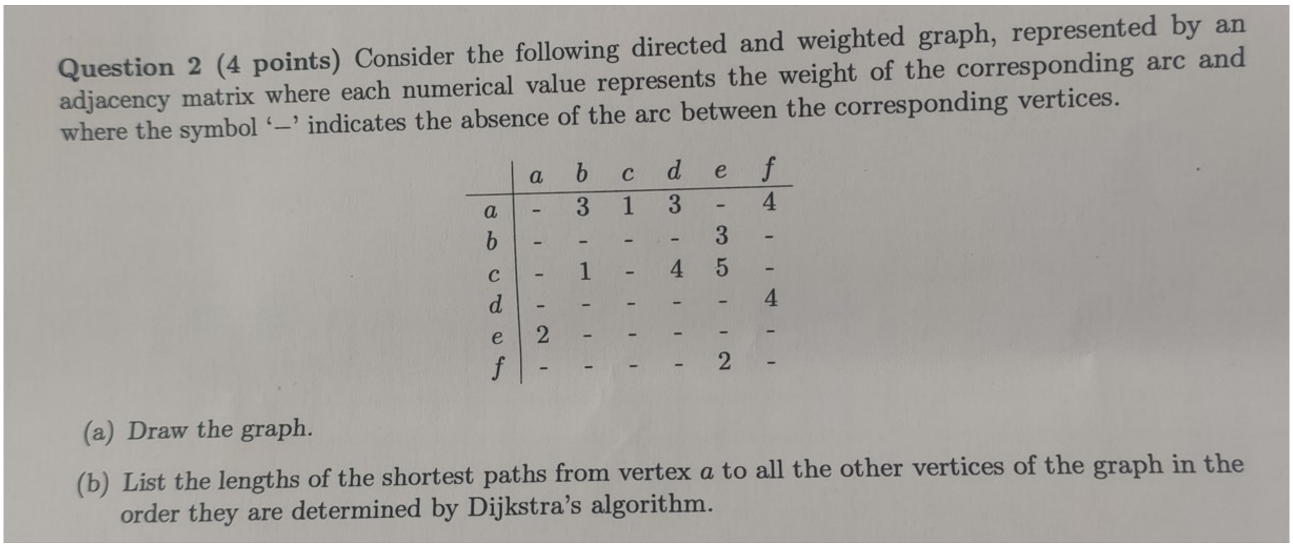


First Part: Theory Questions

Question 1 (4 points) Consider the Union-Find data structure, with the Union operation implemented with union-by-size: show that the complexity of the Find operation is $O(\log n)$, where n is the number of objects in the data structure.

Quoting the theory here (same exercise):
Initially, $depth(x) = 0\ \forall x$. $depth(x)$ can only increase because of a Union in which the root of the tree of x points to another root (depth increases by 1 by construction). This happens only when the tree of x gets merged to a tree of size not smaller (at least as big) $\Rightarrow$ when the depth of x increases, the size of the tree of x at least doubles.

- How many times can this happen?
    - $\leq \log_2 n$ times (at most)
        - therefore the depth of x cannot increase more than $\log_2 n$ times



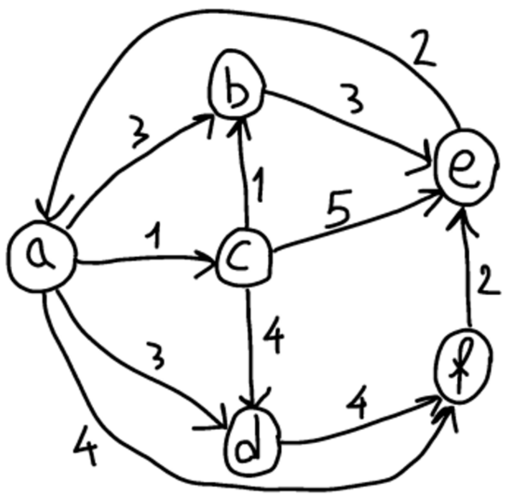
Question 2 (4 points) Consider the following directed and weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding arc and where the symbol '–' indicates the absence of the arc between the corresponding vertices.

	a	b	c	d	e	f
a	-	3	1	3	-	4
b	-	-	-	-	3	-
c	-	1	-	4	5	-
d	-	-	-	-	-	4
e	2	-	-	-	-	-
f	-	-	-	-	2	-

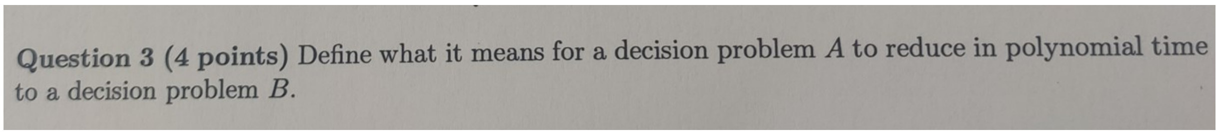
(a) Draw the graph.

(b) List the lengths of the shortest paths from vertex a to all the other vertices of the graph in the order they are determined by Dijkstra's algorithm.

1.



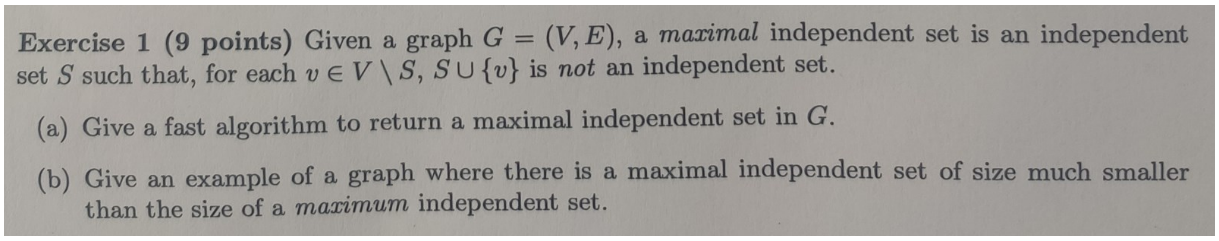
2. $a - c : 1$, $a - b : 2$, $a - d : 3$, $a - f : 4$, $a - e : 5$.
3. $O((m + n) \log n)$.



Question 3 (4 points) Define what it means for a decision problem A to reduce in polynomial time to a decision problem B.

A problem A (pre/post processing has to take at most polynomial time – has to be efficient) reduces in polynomial time to problem B ($A \leq_p B$) if there exist a polynomial time algorithm that transforms an arbitrary input instance a of A into an input instance b of B such that:

1) a is a YES instance of $A \Rightarrow b$ is a YES instance of B
2) b is a YES instance of $B \Rightarrow a$ is a YES instance of A



Exercise 1 (9 points) Given a graph $G = (V, E)$, a *maximal* independent set is an independent set S such that, for each $v \in V \setminus S$, $S \cup \{v\}$ is *not* an independent set.

(a) Give a fast algorithm to return a maximal independent set in G.

(b) Give an example of a graph where there is a maximal independent set of size much smaller than the size of a *maximum* independent set.

a) A fast algorithm to return a maximal independent set in the given graph $G = (V, E)$ can be the following greedy approach:

- Start with an empty set S.
- Iterate through the vertices v in V.
- For each vertex v, if adding v to S does not violate the independence condition (i.e., $S \cup \{v\}$ is an independent set), then add v to S.
- Return the set S as the maximal independent set.

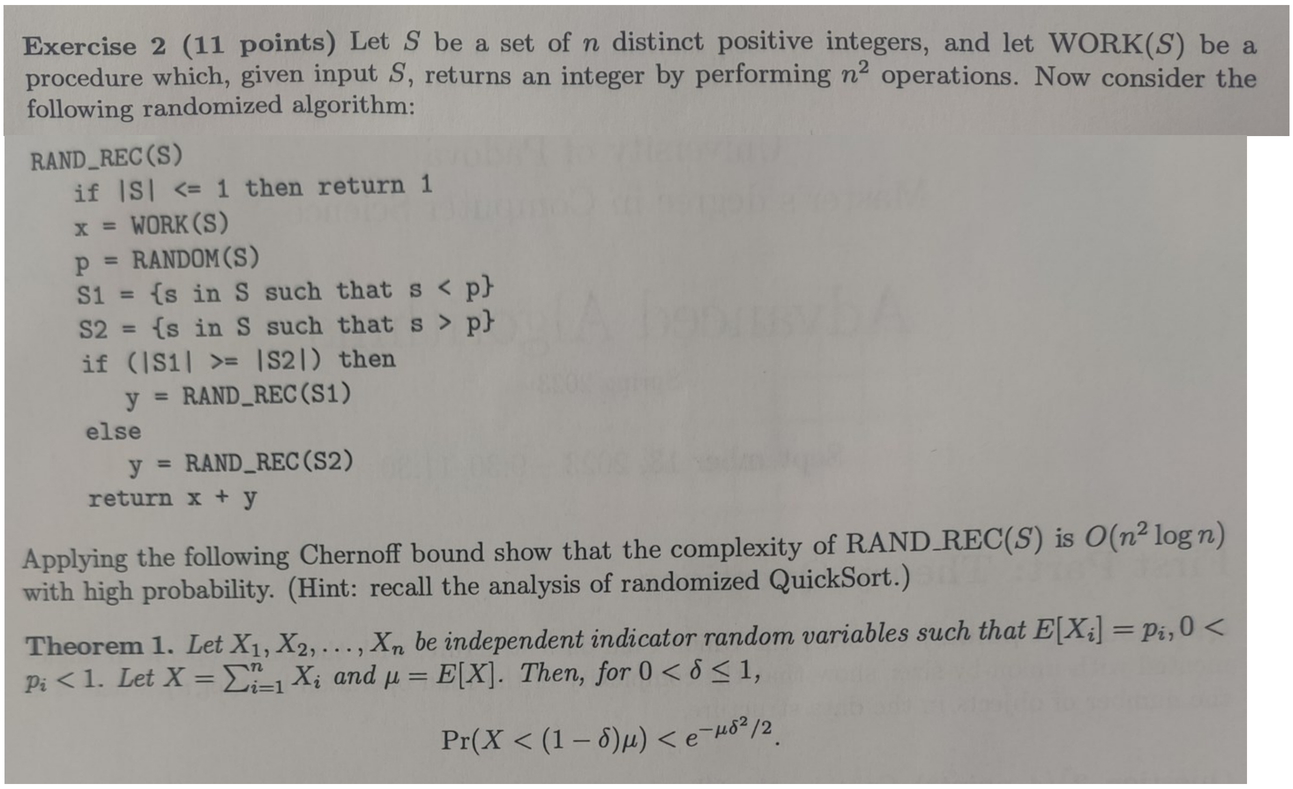
This greedy algorithm has a time complexity of $O(V + E)$ since it iterates through all vertices and checks the independence condition for each vertex, which can be done in $O(deg(v))$ time. Another approach I was suggested would be to run DFS, and checking connected components, if there are as many components as the number of vertices in $V \setminus S$ then it's a maximal independent set

b) Recall "maximal" cannot be increased and it's different from maximum. An example of the concept is a star graph with one central node connected to n leaf nodes.

- The *maximum* independent set would include all the leaf nodes, because none of them are connected to each other. So, if you have a star graph with n leaf nodes, the maximum independent set would be of size n
- However, a *maximal* independent set could be just the central node. Once you include the central node in the set, you can't include any of the leaf nodes because they are all connected to the central node. So, this maximal independent set would be of size 1, which is much smaller than the maximum independent set

Consider as simple example the star graph:

- A maximal independent set can be just the center alone. This set is maximal because adding any vertex to the set formed only by the central vertex would violate the independence property
- A maximum independent set contains all of the other vertices instead, so it includes all the outer vertices and none of the edges within the set violate the independence property



Exercise 2 (11 points) Let S be a set of n distinct positive integers, and let WORK(S) be a procedure which, given input S, returns an integer by performing n^2 operations. Now consider the following randomized algorithm:

```
RAND_REC(S)
  if |S| <= 1 then return 1
  x = WORK(S)
  p = RANDOM(S)
  S1 = {s in S such that s < p}
  S2 = {s in S such that s > p}
  if (|S1| >= |S2|) then
     y = RAND_REC(S1)
  else
     y = RAND_REC(S2)
  return x + y
```

Applying the following Chernoff bound show that the complexity of RAND_REC(S) is $O(n^2 \log n)$ with high probability. (Hint: recall the analysis of randomized QuickSort.)

Theorem 1. *Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that $E[X_i] = p_i, 0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then, for $0 < \delta \leq 1$,*

$$\Pr(X < (1-\delta)\mu) < e^{-\mu\delta^2/2}.$$

Recalling the analysis of Randomized QuickSort: the event E can be characterized as "in the first $l = \log_{\frac{4}{3}}(n)$ nodes of P there have been $< \log_{\frac{4}{3}}(n^2)$ lucky choices". We are studying this specific event:

- $X_i, 1 \leq i \leq l = \log_{\frac{4}{3}}(n^2)$
- $X_i = 1$ if at the i^{th} vertex of P there is a lucky choice of the pivot
- $\Pr(X_i = 1) = \frac{1}{2}\ \forall i$
- X_i are independent

We want the probability of $P\left(\sum_{i=1}^{l} X_i < \log_{\frac{4}{3}}(n^2)\right)$ to bound $X = \sum_{i=1}^{l} X_i$. Given $X = \sum_{i=1}^{l} X_i$, its expected value is as follows:

$$\mu = E[X] = E[\sum_{i=1}^{l} X_i] = \sum_{i=1}^{l} E[X_i] = \sum_{i=1}^{l} \frac{1}{2} = \frac{1}{2} * l = \frac{1}{2}\log_{\frac{4}{3}}(n^2)$$

Now, let's apply the following Chernoff bound (the first):

$$\Pr(X < (1-\delta)\mu) < e^{\frac{-\mu\delta^2}{2}}, 0 < \delta \leq 1$$

$$\downarrow$$

$$(1-\delta)\mu = \log_{\frac{4}{3}}(n^2)$$

$$(1-\delta)\frac{1}{2}\log_{\frac{4}{3}}(n^2) = \log_{\frac{4}{3}}(n^2)$$

$$(1-\delta)\log_{\frac{4}{3}}(n^2) = 2\log_{\frac{4}{3}}(n^2)$$

$$(1-\delta) = 2$$

$$-\delta = 1$$

$$\delta = -1$$

We then apply the Chernoff lemma as follows:

$$\Pr\left(X < \log_{\frac{4}{3}}(n)\right) < e^{-\log\left(\frac{4}{3}\right)(n^2)}$$

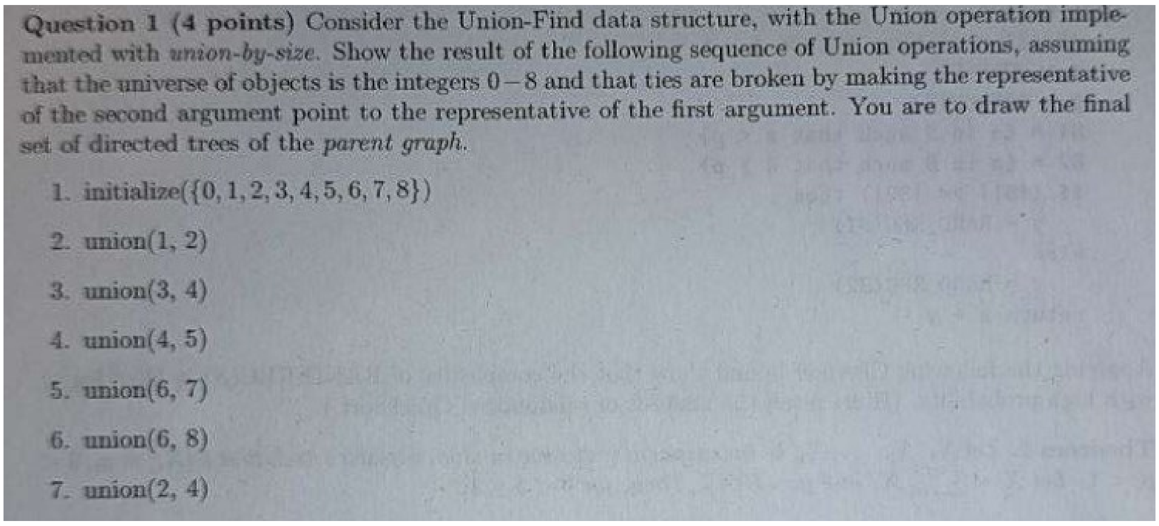
$$= e^{-\frac{ln(n^2)}{ln\left(\frac{4}{3}\right)}}$$

$$= \left(e^{-ln(n^2)}\right)^{\frac{1}{\ln\left(\frac{4}{3}\right)}}$$

$$= \left(\frac{1}{n^2}\right)^{\frac{1}{\ln\left(\frac{4}{3}\right)} \simeq 3{,}47} = \left(\frac{1}{n^2}\right)^3 =$$

$$= \left(\frac{1}{n^2}\right)^3 = (n^{-2})^3 = n^{-6} = \frac{1}{n^6}$$

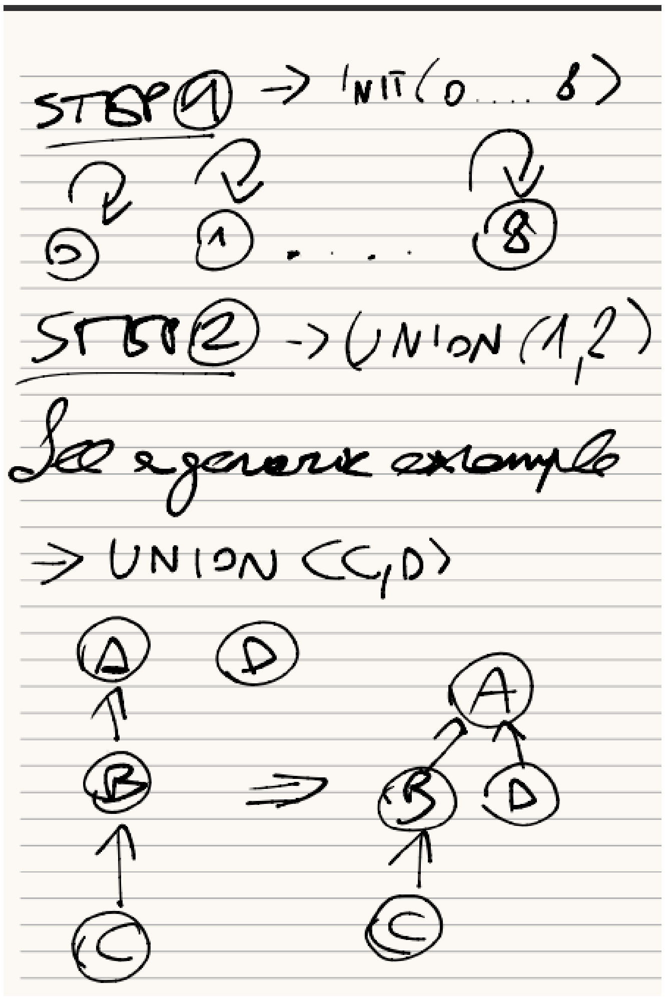
14.10 EXAM OF 02-02-2024

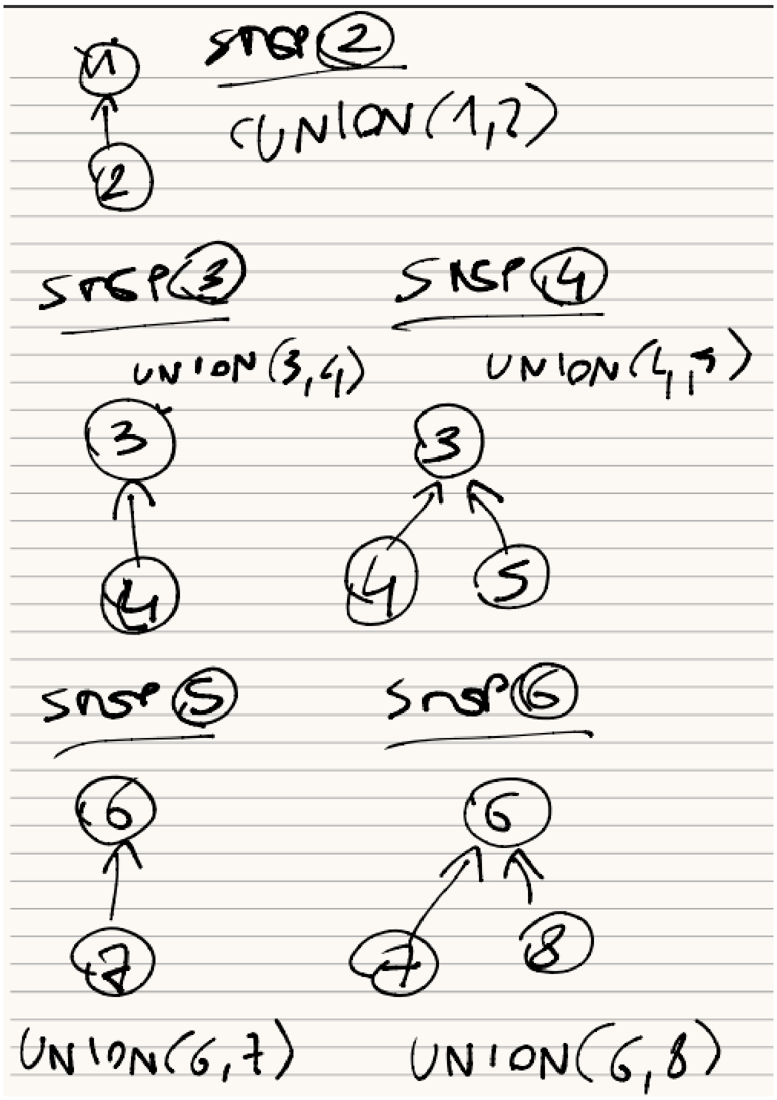


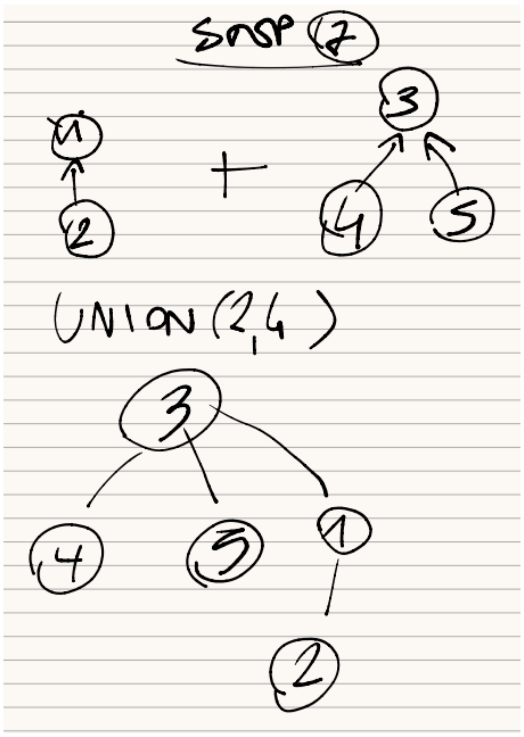
Question 1 (4 points) Consider the Union-Find data structure, with the Union operation implemented with *union-by-size*. Show the result of the following sequence of Union operations, assuming that the universe of objects is the integers 0 – 8 and that ties are broken by making the representative of the second argument point to the representative of the first argument. You are to draw the final set of directed trees of the *parent graph*.

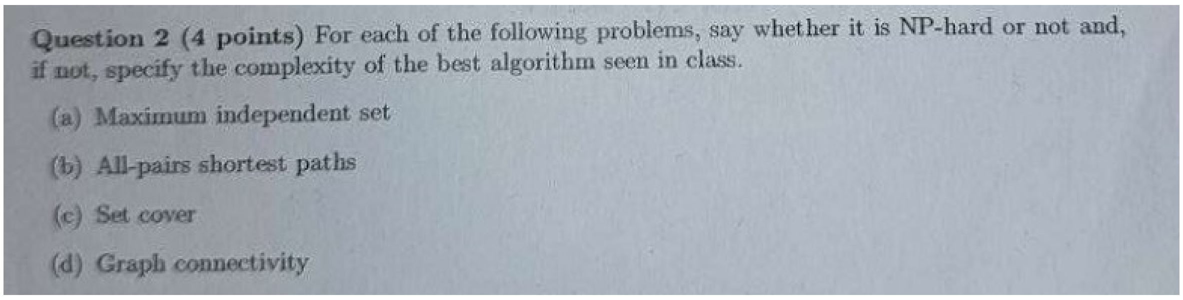
1. initialize({0, 1, 2, 3, 4, 5, 6, 7, 8})
2. union(1, 2)
3. union(3, 4)
4. union(4, 5)
5. union(6, 7)
6. union(6, 8)
7. union(2, 4)

In Union by Rank(or size), when merging two sets, the root of the tree with the smaller rank (or size) becomes a child of the root of the tree with the larger rank. This keeps the tree shallow.








Question 2 (4 points) For each of the following problems, say whether it is NP-hard or not and, if not, specify the complexity of the best algorithm seen in class.

(a) Maximum independent set

(b) All-pairs shortest paths

(c) Set cover

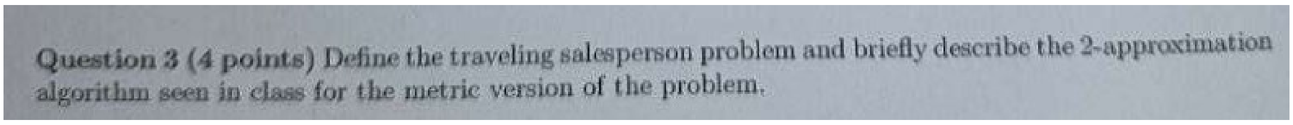
(d) Graph connectivity

(a) NP-Hard

(b) $O(n^3)$

(c) NP-Hard

(d) $O(n * m)$


Question 3 (4 points) Define the traveling salesperson problem and briefly describe the 2-approximation algorithm seen in class for the metric version of the problem.

Given a complete, undirected ($c(u,v) = c(v,u)$ = symmetric) graph $G = (V,E)$ and a function $w: E \to \mathbb{R}^+$, output a tour $T \subseteq E$ (i.e. a cycle that passes through every vertex exactly once) minimizing $\sum_{e \in T} w(e)$. Collectively:

$$T \subseteq E \text{ } minimizing \sum_{e \in T} w(e)$$

We can describe a 2-approximation algorithm where we can find a tour that is no longer than twice the shortest tour. Recall Metric TSP is the problem where all weights respect triangle inequality.

A popular approx-algo to solve this problem is the following:

- Use the most similar problem to Metric TSP → MST (connect all distances with minimal cost employing a fast/simple algorithm = Prim/Kruskal
- Create a list of visited vertices, performing a preorder traversal and visiting every vertex
- Construct an Hamiltonian cycle by skipping any vertex that has already been visited in the walk, effectively "shortcutting" back to the next unvisited city in the preorder list
    - This step is valid and maintains the tour's validity due to the metric property (triangle inequality), which ensures that shortcutting does not increase the overall tour length

To do so, we define the following algorithm:

```
procedure $Approx\_Metric\_TSP(G)$:
    $V = \{v_1, v_2, \dots v_n\}$
    $r = v_1$ $//root\ from\ which\ Prim\ is\ run$
    $T^* = Prim(G, r)$ $//\ compute\ an\ MST\ T\ from\ G\ from\ the\ root\ r$
    $\langle v_{i_1}, v_{i_2}, \dots v_{i_n} \rangle = H' = PREORDER(T^*, r)$
    $//\ lists\ all\ the\ vertices\ in\ the\ tree\ in\ an\ ordered\ fashion\ following\ a\ preorder\ walk$
    return $\langle H', v_{i_1} \rangle \geq H$ $//\ basically, close\ the\ cycle\ and\ return\ it$
```

On the approximation ratio, some comments:

- The weight of the MST is a lower bound for the weight of any tour since any tour containing all cities must at least connect them all
- The shortcut tour created by the preorder walk and subsequent shortcutting can at most double the length of the MST, because each edge of the MST is traversed at most twice (once going to a leaf node and possibly once returning)
- Thus, the total length of the tour obtained by this algorithm is at most twice the length of the optimal tour, giving it a 2-approximation factor

Second Part: Problem Solving

Problem 1 (9 points) Consider Dijkstra's algorithm seen in class, which returns the lengths of the shortest paths from a source vertex to all other vertices in directed graphs with nonnegative weights:

(a) Explain how to modify Dijkstra's algorithm to return the shortest paths themselves (and not just their lengths).

(b) Consider the following algorithm for finding shortest paths in a directed graph where edges may have negative weights: add the same large constant to each edge weight so that all the weights become nonnegative, then run Dijkstra's algorithm and return the shortest paths. Is this a valid method? Either prove that it works (i.e., the returned shortest paths are shortest paths in the original graph), or give a counterexample.

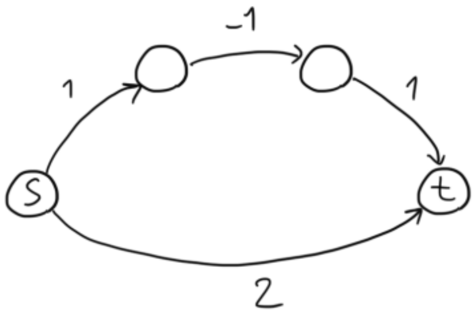
(c) Now let's switch to minimum spanning trees, and do the same: add the same large constant to each edge weight and then run Prim's algorithm. Either prove that the returned solution is a minimum spanning tree of the original graph, or give a counterexample.

Solution:

(a) Associate a pointer *predecessor*(v) to each vertex $v \in V$. When an edge $(v^\star, w^\star)$ is selected in an iteration of the main while loop, assign *predecessor*$(w^\star)$ to $v^\star$. Then, at the end of the algorithm, to reconstruct a shortest path from s to a vertex v, follow the *predecessor* pointers backward from v to s.

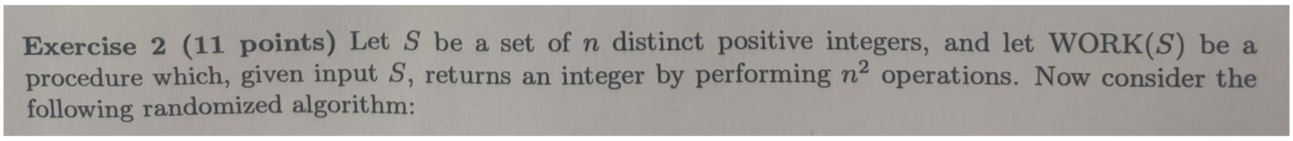
(b) Alas, it does not work. In other words, you cannot reduce the SSSP problem with general edge weights to the special case of nonnegative weights in this way. The problem is that different paths from one vertex to another might not have the same number of edges, hence if we add some number to each edge weight, the lengths of different paths can increase by different amounts, and therefore a shortest path in the new graph might be different than in the original

graph. For example, consider the following graph:

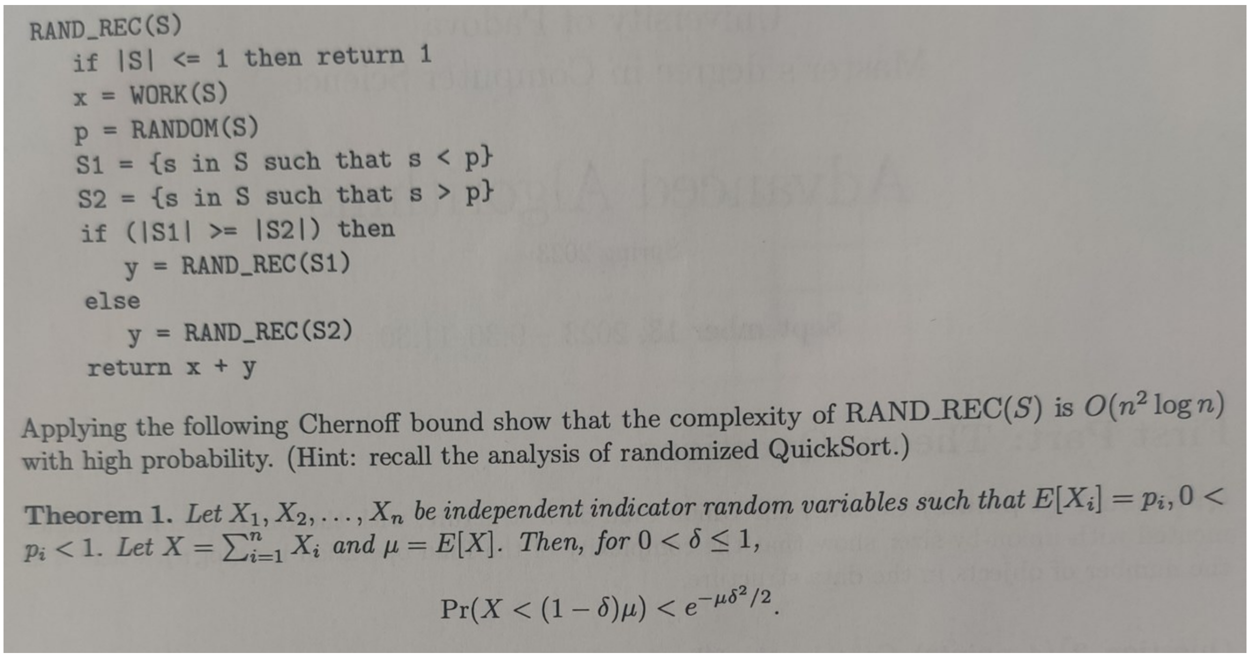


The shortest path from s to t is the 3-hop path on top; however, if we add 1 to every edge's weight, the shortest path from s to t changes, and becomes the direct s-t edge—which has length 3, while the 3-hop path on top has length 4. (We would arrive at the same conclusion no matter which value we choose to force the graph to have nonnegative edge weights.) Moreover, if the original graph has negative cycles, these would not be detected.

(c) For minimum spanning trees this works. In fact, all spanning trees have the same number of edges, hence if we add some number to each edge weight the total weight of different trees increases by the same amount.



Exercise 2 (11 points) Let S be a set of n distinct positive integers, and let WORK(S) be a procedure which, given input S, returns an integer by performing n^2 operations. Now consider the following randomized algorithm:



```
RAND_REC(S)
   if |S| <= 1 then return 1
   x = WORK(S)
   p = RANDOM(S)
   S1 = {s in S such that s < p}
   S2 = {s in S such that s > p}
   if (|S1| >= |S2|) then
      y = RAND_REC(S1)
   else
      y = RAND_REC(S2)
   return x + y
```

Applying the following Chernoff bound show that the complexity of RAND_REC(S) is $O(n^2 \log n)$ with high probability. (Hint: recall the analysis of randomized QuickSort.)

Theorem 1. *Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that $E[X_i] = p_i, 0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then, for $0 < \delta \leq 1$,*

$$\Pr(X < (1-\delta)\mu) < e^{-\mu\delta^2/2}.$$

Recalling the analysis of Randomized QuickSort: the event E can be characterized as "in the first $l = \log_{\frac{4}{3}}(n)$ nodes of P there have been $< \log_{\frac{4}{3}}(n^2)$ lucky choices". We are studying this specific event:

- $X_i, 1 \leq i \leq l = \log_{\frac{4}{3}}(n^2)$
- $X_i = 1$ if at the i^{th} vertex of P there is a lucky choice of the pivot
- $\Pr(X_i = 1) = \frac{1}{2}\ \forall i$
- X_i are independent

We want the probability of $P\left(\sum_{i=1}^{l} X_i < \log_{\frac{4}{3}}(n^2)\right)$ to bound $X = \sum_{i=1}^{l} X_i$. Given $X = \sum_{i=1}^{l} X_i$, its expected value is as follows:

$$\mu = E[X] = E[\sum_{i=1}^{l} X_i] = \sum_{i=1}^{l} E[X_i] = \sum_{i=1}^{l} \frac{1}{2} = \frac{1}{2} * l = \frac{1}{2}\log_{\frac{4}{3}}(n^2)$$

Now, let's apply the following Chernoff bound (the first):

$$\Pr(X < (1-\delta)\mu) < e^{\frac{-\mu\delta^2}{2}}, 0 < \delta \leq 1$$

$$\downarrow$$

$$(1-\delta)\mu = \log_{\frac{4}{3}}(n^2)$$

$$(1-\delta)\frac{1}{2}\log_{\frac{4}{3}}(n^2) = \log_{\frac{4}{3}}(n^2)$$

$$(1-\delta)\log_{\frac{4}{3}}(n^2) = 2\log_{\frac{4}{3}}(n^2)$$

$$(1-\delta) = 2$$

$$-\delta = 1$$

$$\delta = -1$$

We then apply the Chernoff lemma as follows:

$$\Pr\left(X < \log_{\frac{4}{3}}(n)\right) < e^{-\log\left(\frac{4}{3}\right)(n^2)}$$

$$= e^{-\frac{ln(n^2)}{ln\left(\frac{4}{3}\right)}}$$

$$= \left(e^{-ln(n^2)}\right)^{\frac{1}{\ln\left(\frac{4}{3}\right)}}$$

$$= \left(\frac{1}{n^2}\right)^{\frac{1}{\ln\left(\frac{4}{3}\right)} \simeq 3{,}47}$$

$$= \left(\frac{1}{n^2}\right)^3 = (n^{-2})^3 = n^{-6} = \frac{1}{n^6}$$

14.11 Exam of 29-06-2022

Question 1 (6 points) Consider the following weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding edge, and where the symbol '–' indicates the absence of the edge between the corresponding nodes.

	a	b	c	d	e	f
a	-	3	7	-	1	5
b		-	6	8	-	-
c			-	2	9	-
d				-	-	-
e					-	4
f						-

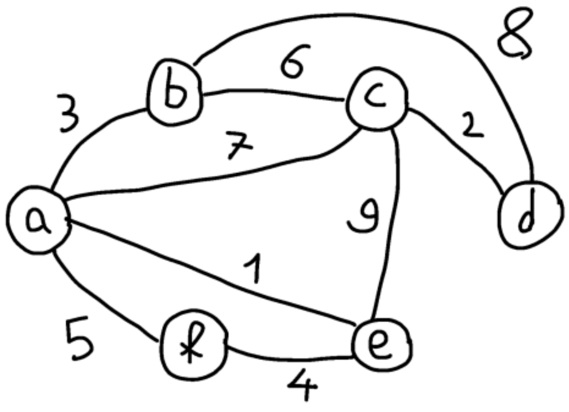
(a) Draw the graph.

(b) List the edges of the minimum spanning tree in the order they are selected by Kruskal's algorithm.

(c) List the edges of the minimum spanning tree in the order they are selected by Prim's algorithm starting at node c.

Solution:

(a)



(b) $(a, e), (c, d), (a, b), (e, f), (b, c)$.

(c) $(c, d), (b, c), (a, b), (a, e), (e, f)$.

Question 2 (7 points) For each of the following problems, say whether it is NP-hard or not and, if not, specify the complexity of the best algorithm seen in class.

(a) Single-source shortest paths

(b) Minimum vertex cover

(c) Connected components

(d) 3SAT

(e) Minimum spanning trees

(f) Metric TSP

(g) Maximum independent set

Solution:

(a) $O((m + n) \log n)$

(b) NP-hard

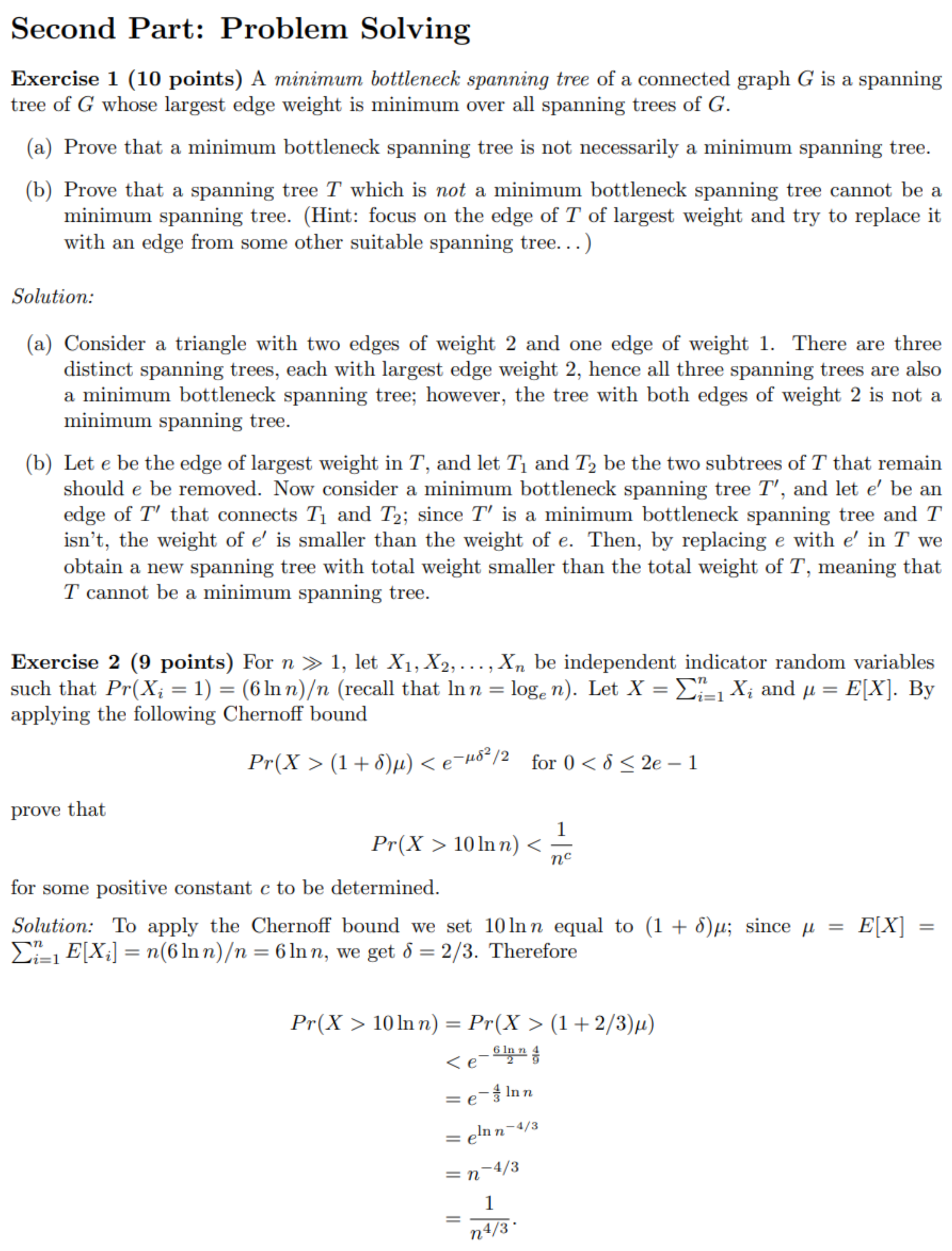
(c) $O(m + n)$

(d) NP-hard

(e) $O(m \log n)$

(f) NP-hard

(g) NP-hard



Second Part: Problem Solving

Exercise 1 (10 points) A *minimum bottleneck spanning tree* of a connected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G.

(a) Prove that a minimum bottleneck spanning tree is not necessarily a minimum spanning tree.

(b) Prove that a spanning tree T which is *not* a minimum bottleneck spanning tree cannot be a minimum spanning tree. (Hint: focus on the edge of T of largest weight and try to replace it with an edge from some other suitable spanning tree...)

Solution:

(a) Consider a triangle with two edges of weight 2 and one edge of weight 1. There are three distinct spanning trees, each with largest edge weight 2, hence all three spanning trees are also a minimum bottleneck spanning tree; however, the tree with both edges of weight 2 is not a minimum spanning tree.

(b) Let e be the edge of largest weight in T, and let T_1 and T_2 be the two subtrees of T that remain should e be removed. Now consider a minimum bottleneck spanning tree T', and let e' be an edge of T' that connects T_1 and T_2; since T' is a minimum bottleneck spanning tree and T isn't, the weight of e' is smaller than the weight of e. Then, by replacing e with e' in T we obtain a new spanning tree with total weight smaller than the total weight of T, meaning that T cannot be a minimum spanning tree.

Exercise 2 (9 points) For $n \gg 1$, let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that $Pr(X_i = 1) = (6 \ln n)/n$ (recall that $\ln n = \log_e n$). Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. By applying the following Chernoff bound

$$Pr(X > (1+\delta)\mu) < e^{-\mu\delta^2/2} \quad \text{for } 0 < \delta \le 2e - 1$$

prove that

$$Pr(X > 10 \ln n) < \frac{1}{n^c}$$

for some positive constant c to be determined.

Solution: To apply the Chernoff bound we set $10 \ln n$ equal to $(1+\delta)\mu$; since $\mu = E[X] = \sum_{i=1}^{n} E[X_i] = n(6 \ln n)/n = 6 \ln n$, we get $\delta = 2/3$. Therefore

$$\begin{aligned} Pr(X > 10 \ln n) &= Pr(X > (1 + 2/3)\mu) \\ &< e^{-\frac{6 \ln n}{2}\frac{4}{9}} \\ &= e^{-\frac{4}{3}\ln n} \\ &= e^{\ln n^{-4/3}} \\ &= n^{-4/3} \\ &= \frac{1}{n^{4/3}}. \end{aligned}$$

To expand a bit on the solution:

Here we have already $\Pr(X_i = 1) = \frac{6\ln(n)}{n}$ and we need to find $\mu = E[X] = E[\sum_{i=1}^{n} X_i] = n * \frac{6\ln(n)}{n} = 6\ln(n)$. Now, we find δ:

$$(1+\delta)\mu = 10\ln(n)$$

$$(1+\delta)6\ln(n) = 10\ln(n)$$

$$(1+\delta) = \frac{5}{3}$$

$$\delta = \frac{2}{3}$$

Now, we use the bound:

$$\Pr(X > 10\ln(n)) = \Pr(X > (1+\delta)\mu) = \Pr\left(X > \left(1+\frac{2}{3}\right)\mu\right)$$

$$< e^{-\frac{6\ln(n)}{2} * \frac{4}{9}}$$

$$< e^{-\frac{4}{3}\ln(n)}$$

$$< e^{\ln(n)^{-\frac{4}{3}}}$$

$$< n^{-\frac{4}{3}}$$

$$= \frac{1}{n^{\frac{4}{3}}}$$

Which is then verified for the constant $c = \frac{4}{3}, c > 0$ as showed.

14.12 Exam of 29-06-2020

(Note: basically this entire file was translated into English and put as example of exam exercises every year since 2021-2022, since when Scquizzato became the only teacher of this course)

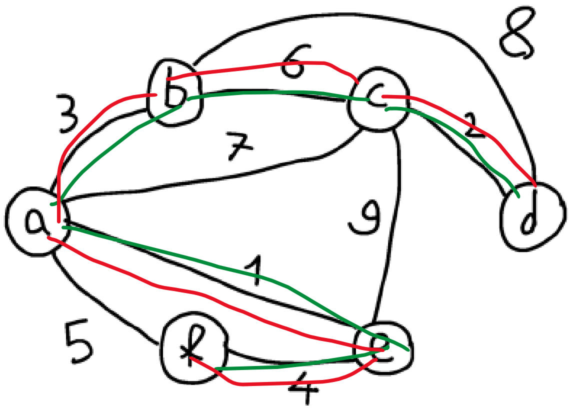
Question 1 (6 points) Consider the following weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding edge, and where the symbol '–' indicates the absence of the edge between the corresponding nodes.

	a	b	c	d	e	f
a	-	3	7	-	1	5
b		-	6	8	-	-
c			-	2	9	-
d				-	-	-
e					-	4
f						-

1. Draw the graph.
2. List the edges of the minimum spanning tree in the order they are selected by Kruskal's algorithm.
3. List the edges of the minimum spanning tree in the order they are selected by Prim's algorithm starting at node a.

Solution:

1.



2. $(a, e), (c, d), (a, b), (e, f), (b, c)$.
3. $(a, e), (a, b), (e, f), (b, c), (c, d)$.

Question 2 (7 points) With reference to the problem of the minimum vertex cover, which in class we have 2-approximated by computing a maximal matching:

1. Give the definition of vertex cover of a graph.
2. Give the definition of matching of a graph.
3. Find a maximal matching in the graph of Question 1.

Solution:

1. A vertex cover of a graph is a set of vertices that includes at least one endpoint of every edge of the graph.
2. A matching of a graph is a set of edges without common vertices.
3. For example $\{(a, b), (c, d), (e, f)\}$ or $\{(a, c), (b, d), (e, f)\}$.

Second Part: Problem Solving

Exercise 1 (11 points) Given a set S of n integers and an additional integer t, assume that $\forall s \in S, 0 \leq s \leq t$. Consider the optimization problem where the set of feasible solutions is

$$\{S' \subseteq S \,\text{such that} \sum_{s \in S'} s \leq t\},$$

the cost of a feasible solution S' is $c(S') = \sum_{s \in S'} s$, and the goal is to compute the maximum cost among all the costs of the feasible solutions.

1. Design a simple 2-approximation polynomial-time algorithm for this problem. (Hint: consider a *descending* ordering of the values in S, and then do a single pass over such values.)
2. Prove that such algorithm is a 2-approximation algorithm.

Solution:

1.
```
APPROX_SS(S, t)
{s_1, s_2,..., s_n} <- SORT-DECREASING(S)
sum = s_1
for i = 2 to n do
   if sum + s_i <= t then
      sum = sum + s_i
   else
      return sum
return sum
```

2. First of all we observe that, since sum is initialized to $s_1 \leq t$ and a value s_i is added to sum only if $sum + s_i \leq t$, the returned value is always the cost of a feasible solution. If $s^\star$ denotes the maximum cost, we now need to prove that $s^\star/sum \leq 2$.

   case 1 The algorithm returns out of the for loop: hence $sum = \sum_{s \in S} s \leq t$, that is $sum = s^\star$, and thus $s^\star/sum = 1 \leq 2$.

   case 2 The algorithm returns from inside the for loop: hence there exists and index i' such that $sum + s_{i'} > t$. Observe that
   $$s_{i'} < s_1 \leq sum,$$
   and hence
   $$2 \cdot sum > sum + s_{i'} > t,$$
   that is
   $$sum > \frac{t}{2} \geq \frac{s^\star}{2}.$$

Exercise 2 (9 points) Let $X_1, X_2, \ldots, X_n$ be independent indicator random variables such that $Pr(X_i = 1) = 1/(4e)$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. By applying the following Chernoff bound, which holds for every $\delta > 0$,

$$Pr(X > (1+\delta)\mu) < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$$

prove that

$$Pr(X > n/2) < \frac{1}{(\sqrt{2})^n}.$$

Solution: To apply the Chernoff bound we set $n/2$ equal to $(1+\delta)\mu$; since $\mu = E[X] = \sum_{i=1}^{n} E[X_i] = n/(4e)$, we get $\delta = 2e - 1$. Therefore

$$\begin{aligned} Pr(X > n/2) &= Pr(X > (1 + 2e - 1)\mu) \\ &< \left(\frac{e^{2e-1}}{(2e)^{2e}}\right)^{n/(4e)} \\ &< 1/(2^{2e/4e})^n \\ &= \left(1/\sqrt{2}\right)^n. \end{aligned}$$

Why this computation? The following explains it exactly:

So, here, we have to first consider X_i. We know they are independent. Now, we simply need to find the expected value. We already have here the probabilty of success given by $\Pr(X_i = 1) = \frac{1}{4e}$. This applies for all$n$ events since they are independent so:

$$\mu = E[X] = E[\sum_{i=1}^{n} E[X_i] = n * \frac{1}{4e} = \frac{n}{4e}$$

Now, we find δ and this is done according to value we have to bound, given by the exercise or explicitly told here like $\frac{n}{2}$:

$$(1+\delta)\mu = \frac{n}{2}$$

$$(1+\delta)\frac{n}{4e} = \frac{n}{2}$$

$$(1+\delta)\frac{n}{2e} = n$$

$$(1+\delta)n = 2e(n)$$

$$(1+\delta) = 2e$$

$$\delta = 2e - 1$$

Now that we found δ, let's plug it in back in the original bound:

$$\Pr(1+\delta)\,\mu < \left(\frac{(e^{\delta})}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

$$= \Pr\left(X > (1+2e-1)\frac{n}{4e}\right) \leq$$

$$\leq \left(\frac{e^{2e-1}}{(1+2e-1)^{(1+2e-1)}}\right)^{\frac{n}{4e}}$$

$$\leq \left(\frac{e^{2e-1}}{(2e)^{(2e)}}\right)^{\frac{n}{4e}}$$

$$\leq \left(\frac{\cancel{e^{2e}} * e^{-1}}{2^{2e} * (\cancel{e^{2e}})}\right)^{\frac{n}{4e}}$$

$$\leq \left(\frac{1}{e}\right)^{\frac{n}{4e}} * \left(\frac{1}{2^{2e}}\right)^{\frac{n}{4e}}$$

$$\leq \left(\frac{1}{e^{-4e}}\right)^{n} * \left(\frac{1}{2^{\frac{2e}{4e}}}\right)^{n}$$

$$\leq \left(\frac{1}{e^{-4e}}\right)^{n} * \left(\frac{1}{2^{\frac{1}{2}}}\right)^{n}$$

$$\leq \left(\frac{1}{e^{-4e}}\right)^n * \left(\frac{1}{\sqrt{2}}\right)^n$$

To infinity, it dominates the second factor, so we'd have $\left(\frac{1}{\sqrt{2}}\right)^n$

14.13 EXAM OF 15-07-2020

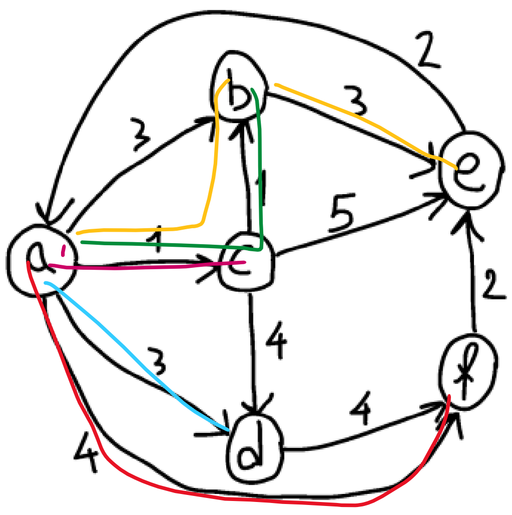
Domanda 1 (6 punti) Si consideri il seguente grafo pesato e diretto, rappresentato tramite una matrice di adiacenza dove ogni valore numerico rappresenta il peso dell'arco corrispondente, e dove il simbolo '−' indica l'assenza dell'arco tra i nodi corrispondenti.

	a	b	c	d	e	f
a	-	3	1	3	-	4
b	-	-	-	-	3	-
c	-	1	-	4	5	-
d	-	-	-	-	-	4
e	2	-	-	-	-	-
f	-	-	-	-	2	-

1. Disegnare il grafo.
2. Elencare le lunghezze dei cammini più brevi tra il nodo a e tutti gli altri nodi del grafo, nell'ordine in cui queste sono determinate dall'algoritmo di Dijkstra.
3. Qual'è la complessità dell'algoritmo di Dijkstra quando implementato con heap?

Soluzione:

1.



2. $a-c:1$, $a-b:2$, $a-d:3$, $a-f:4$, $a-e:5$.
3. $O((m+n)\log n)$.

Dijkstra: finds the shortest paths to <u>all</u> vertices starting from a vertex

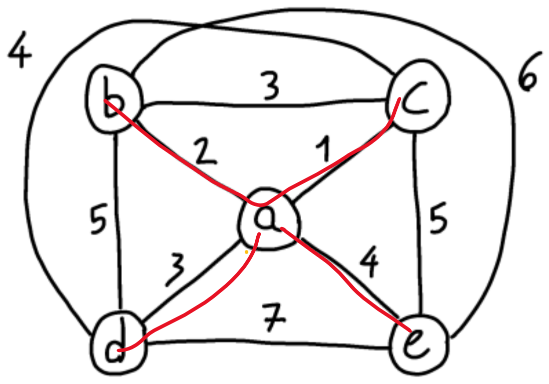
Here, $Approx\_T\_TSP$ is the older name of $Approx\_Metric\_TSP$, so it's useful to know the below graph.

Domanda 2 (6 punti) Si consideri il seguente grafo completo pesato, rappresentato tramite una matrice di adiacenza dove ogni valore numerico rappresenta il peso del lato corrispondente.

	a	b	c	d	e
a	-	2	1	3	4
b		-	3	5	6
c			-	4	5
d				-	7
e					-

1. Disegnare il grafo.
2. Elencare i lati del minimum spanning tree nell'ordine in cui sono selezionati dall'algoritmo di Prim a partire dal nodo a.
3. Siccome i pesi dei lati soddisfano la disuguaglianza triangolare, si può applicare l'algoritmo di 2-approssimazione APPROX_T_TSP visto in classe per il TSP. Considerando i nodi ordinati secondo l'ordine alfabetico, fornire l'output dell'algoritmo APPROX_T_TSP quando eseguito su questo grafo.

Soluzione:

1. 

2. $(a, c), (a, b), (a, d), (a, e)$.
3. Considerando i nodi ordinati secondo l'ordine alfabetico, l'algoritmo APPROX_T_TSP esegue l'algoritmo di Prim a partire dal nodo a. Un possibile output è quindi a, c, b, d, e, a.

Seconda Parte: risoluzione di problemi

Esercizio 1 (10 punti) Un grafo $G = (V, E)$ si dice *bipartito* se l'insieme di vertici V può essere partizionato in due sottoinsiemi X e Y tali che ogni lato di E incide su un vertice di X e uno di Y. Progettare e analizzare un algoritmo efficiente che determini se un grafo G è bipartito.

Soluzione: Si osservi che G è bipartito se e solo se ciascuna sua componente connessa è bipartita. Basta quindi controllare che ogni componente connessa sia bipartita.

Si consideri quindi un'arbitraria componente connessa. Un possibile algoritmo si basa sull'uso della BFS:

1. Eseguire una BFS da un vertice qualsiasi della componente connessa
2. Colorare i vertici del BFS tree blu se sono in un livello pari, rosso se sono in un livello dispari (quindi, X corrisponde a blu e Y a rosso, o viceversa)
3. Controllare ogni lato per vedere se ne esiste uno che incide su due vertici dello stesso colore: se sì, allora return FALSE, altrimenti return TRUE

Correttezza dell'algoritmo: se non esiste nessun lato che incide su due vertici dello stesso colore, allora si ha una corretta bipartizione e quindi la componente è bipartita. Altrimenti, la componente non può essere bipartita: se lo fosse, non avrebbe lati che incidono su vertici dello stesso colore.

Per quanto riguarda la complessità, l'algoritmo equivale a una visita di tutto il grafo che, come visto a lezione, ha complessità $O(m + n)$.

Esercizio 2 (10 punti) $m = 6n \ln n$ jobs vengono assegnati in modo casuale a n processori. (Nota: si ricordi che $\ln x = \log_e x$.) Si consideri un certo processore p, e si dimostri che, con alta probabilità nel parametro n, il processore p non riceve più di $12 \ln n$ jobs. (Suggerimento: definire una opportuna variabile indicatore per ogni job, e applicare il seguente bound di Chernoff.)

Theorem 1. *Siano $X_1, X_2, \ldots, X_n$ variabili indicatore indipendenti con $E[X_i] = p_i, 0 < p_i < 1$. Sia $X = \sum_{i=1}^{n} X_i$ e $\mu = E[X]$. Allora, per ogni $0 < \delta \le 1$,*

$$Pr(X > (1 + \delta)\mu) \le e^{-\frac{\delta^2 \mu}{3}}.$$

Soluzione: Sia X_i, con $i = 1, 2, \ldots, 6n \ln n$, una variabile indicatore che vale 1 se l'i-esimo job viene assegnato al processore p. Ovviamente, $Pr(X_i = 1) = 1/n$, e le X_i sono indipendenti. Il numero di job ricevuti dal processore p è quindi $X = \sum_{i=1}^{6n \ln n} X_i$. Per applicare il bound di Chernoff poniamo $12 \ln n$ uguale a $(1+\delta)\mu$; siccome $\mu = E[X] = \sum_{i=1}^{6n \ln n} E[X_i] = (6n \ln n)/n = 6 \ln n$, otteniamo $\delta = 1$. Quindi

$$Pr(X > 12 \ln n) \le e^{-\frac{6 \ln n}{3}} = 1/n^2,$$

cioè

$$Pr(X \le 12 \ln n) > 1 - 1/n^2.$$

To expand properly on the solution – also, if you see some exams ago, we see that this is basically the same thing with bins, but also union bound needs to be applied here.

Let $X_1 = 1, 2, \dots m$ (with $m = 6nln(n)$ jobs and $X_i = 1$ when the i^{th} job gets assigned to processor p. Here, $\Pr(X_i = 1) = \frac{1}{n}$ with X_i independent between each other. The number of jobs received by the processor p is then $X = \sum_{i=1}^{m} X_i$ given it holds for each processor.

Now, find $\mu = E[X] = \sum_{i=1}^{m} E[X_i] = m * \frac{1}{n} = \frac{6n(\ln(n))}{n} = 6\ln(n)$

Then, we find δ:

To apply the Chernoff bound, we set $12\ln(n)$ equal to $(1 + \delta)$ so:

$$(1 + \delta)\mu = 12\ln(n)$$

$$(1 + \delta)6\ln(n) = 12\ln(n)$$

$$(1 + \delta) = 2$$

$$\delta = 1$$

Now, we apply the bound:

$$\Pr(X > (1 + \delta)\mu) \le e^{\frac{-\mu\delta^2}{3}}$$

$$\Pr(X > 1 + 1)\,6\ln(n)) \le e^{-\frac{6\ln(n)}{3}}$$

$$\le e^{-2\ln(n)}$$

Recall the property of exponentials and logarithms there, so:

$$\le e^{\ln(n^{-2})}$$

Recall from the exercise hint that $\ln(n) = \log_e(n)$

So, we have:

$$e^{\ln(\mathrm{n}^{-2})} = \frac{1}{n^2}$$

as the exercise wanted. We showed with high probability the bin with maximum load containing at most $12\ln(n)$ jobs. We applied this for *one* bin, so we have to use now the union bound; simply use the previous result multiplying by all jobs, so $m = \frac{n}{6n(\ln(n))}$: $\frac{n}{6\ln(n)} * \frac{1}{n^2} = \frac{1}{6nln(n)}$

To characterize the *no job will exceed*, use the complement event → $1 - \frac{1}{6nln(n)} = 1 - o\left(\frac{1}{n}\right)$

14.14 Exam of 31-08-2020

(Note: some exercises of this one was translated into English and put as examples of exam exercises every year of this course since 2021-2022, when only Scquizzato is teaching this course – I put here the English versions of said exercises)

Question 1

Consider the following weighted graph, represented by an adjacency matrix where each numerical value represents the weight of the corresponding edge, and where the symbol '–' indicates the absence of the edge between the corresponding nodes.

	a	b	c	d	e	f
a	-	3	7	-	1	5
b		-	6	8	-	-
c			-	2	9	-
d				-	-	-
e					-	4
f						-

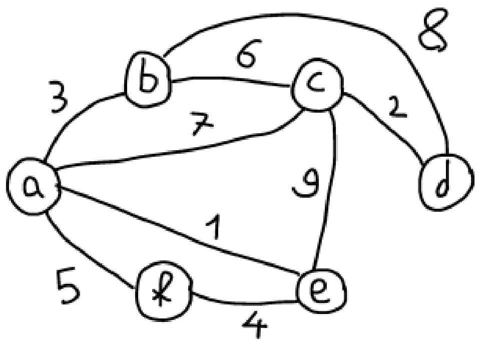
(a) Draw the graph.

(b) List the edges of the minimum spanning tree in the order they are selected by Kruskal's algorithm.

(c) List the edges of the minimum spanning tree in the order they are selected by Prim's algorithm starting at node a.

Solution:

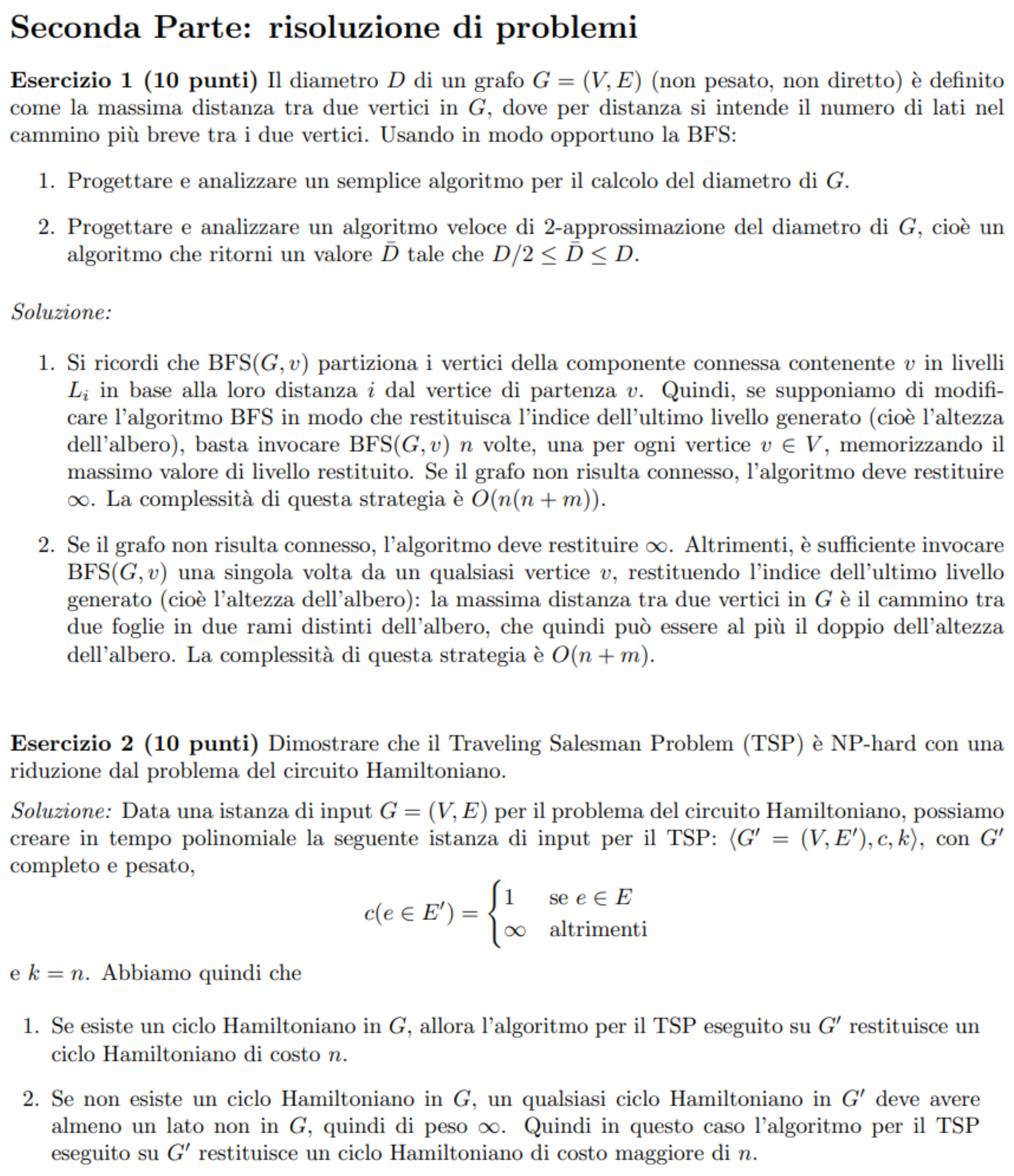
(a)



Show that a graph with n vertices and with a minimum cut of size t has at least $tn/2$ edges.

Solution: By definition of minimum cut we have $d(v) \geq t$ for any vertex $v \in V$, where $d(v)$ denotes the degree of v. Then, summing up over all the n vertices we obtain $\sum_{v \in V} d(v) \geq tn$. The claim follows by observing that $\sum_{v \in V} d(v) = 2m$.

Second property holds thanks to the Handshaking lemma, shown in the beginning of this course as graph property.



Seconda Parte: risoluzione di problemi

Esercizio 1 (10 punti) Il diametro D di un grafo $G = (V, E)$ (non pesato, non diretto) è definito come la massima distanza tra due vertici in G, dove per distanza si intende il numero di lati nel cammino più breve tra i due vertici. Usando in modo opportuno la BFS:

1. Progettare e analizzare un semplice algoritmo per il calcolo del diametro di G.
2. Progettare e analizzare un algoritmo veloce di 2-approssimazione del diametro di G, cioè un algoritmo che ritorni un valore $\bar{D}$ tale che $D/2 \leq \bar{D} \leq D$.

Soluzione:

1. Si ricordi che BFS(G, v) partiziona i vertici della componente connessa contenente v in livelli L_i in base alla loro distanza i dal vertice di partenza v. Quindi, se supponiamo di modificare l'algoritmo BFS in modo che restituisca l'indice dell'ultimo livello generato (cioè l'altezza dell'albero), basta invocare BFS(G, v) n volte, una per ogni vertice $v \in V$, memorizzando il massimo valore di livello restituito. Se il grafo non risulta connesso, l'algoritmo deve restituire ∞. La complessità di questa strategia è $O(n(n + m))$.
2. Se il grafo non risulta connesso, l'algoritmo deve restituire ∞. Altrimenti, è sufficiente invocare BFS(G, v) una singola volta da un qualsiasi vertice v, restituendo l'indice dell'ultimo livello generato (cioè l'altezza dell'albero): la massima distanza tra due vertici in G è il cammino tra due foglie in due rami distinti dell'albero, che quindi può essere al più il doppio dell'altezza dell'albero. La complessità di questa strategia è $O(n + m)$.

Esercizio 2 (10 punti) Dimostrare che il Traveling Salesman Problem (TSP) è NP-hard con una riduzione dal problema del circuito Hamiltoniano.

Soluzione: Data una istanza di input $G = (V, E)$ per il problema del circuito Hamiltoniano, possiamo creare in tempo polinomiale la seguente istanza di input per il TSP: $\langle G' = (V, E'), c, k \rangle$, con G' completo e pesato,

$$c(e \in E') = \begin{cases} 1 & \text{se } e \in E \\ \infty & \text{altrimenti} \end{cases}$$

e $k = n$. Abbiamo quindi che

1. Se esiste un ciclo Hamiltoniano in G, allora l'algoritmo per il TSP eseguito su G' restituisce un ciclo Hamiltoniano di costo n.
2. Se non esiste un ciclo Hamiltoniano in G, un qualsiasi ciclo Hamiltoniano in G' deve avere almeno un lato non in G, quindi di peso ∞. Quindi in questo caso l'algoritmo per il TSP eseguito su G' restituisce un ciclo Hamiltoniano di costo maggiore di n.

First exercise is a variant found within the CLRS book, second one comes straight from the theory.

14.15 Older Exercises

This includes exercises found from old lessons and/or notes in italian and represents a complete translation or solution for all of them. At last to properly conclude this file.

1. Given a tree with n nodes has $m = n - 1$ edges, show that a connected graph with n nodes has $m \geq n - 1$ edges.

Solution

Reason by contradiction and suppose we have a connected graph G with n nodes and fewer than $n - 1$ edges. Since G is connected, there exists at least one spanning tree (subgraph with all the vertices as the original graph). So, with m' the number of edges in T, by definition of a tree we would have $m' = n - 1$.

Considering we have a T as subgraph of G, it would have as number of edges a number greater than or equal to the number of edges in T.

Let m be the number of edges in G. Then, $m \geq m' = n - 1$. This contradicts our initial assumption that G has fewer than $n - 1$ edges.

Therefore, a connected graph with n nodes must have at least $n - 1$ edges. This proof also shows that a connected graph with n nodes and exactly $n - 1$ edges is a tree.

In summary:

- A tree with n nodes has exactly $n - 1$ edges.
- A connected graph with n nodes must contain at least one spanning tree.
- A spanning tree of a graph with n nodes has $n - 1$ edges.
- Therefore, a connected graph with n nodes must have at least $n - 1$ edges.

2. Consider a labyrinth L that has a single entry point s and within which is hidden the terrible Minotaur. Engineer Theseus must enter L, find the Minotaur, kill it, and exit L. Find an appropriate representation of the labyrinth as a graph and show how, by exploiting the DFS algorithm, ing. Theseus can successfully accomplish his mission. Assume that the labyrinth is connected, in the sense that every point in it can be reached from s.

The following peculiarities can be added to the nodes of the graph-maze $L = (V, E)$:

- $s \in L$ and the gateway node, i.e., the entry and exit node from the maze L
- $\forall\, v \in L$:
    a. $L_V[v].hasMinotaur = true$ if there is Minotaur at that node
    b. $L_V[v].hasMinotaur = false$ otherwise

$\exists!\, v \in L: L_V[v].hasMinotaur = true$, which means there is only a Minotaur inside of the labyrinth L.

Solution

```
$MinotaurDFS(G, v)$
    $LV\ [v].ID \leftarrow 1$
    $if\ (LV\ [v].hasMinotaur = true)$
        $killTheMinotaur()$
        return $true$
    forall $e \in G.incidentEdges(v)$: do
        $if(LE[e].label\ =\ null)\ then$
            $w \leftarrow G.opposite(v, e)$
        $if(LV\ [w].ID\ =\ 0)\ then$
            $LE[e].label\ \leftarrow\ ARIADNE'S\ ROPE$
            $killed = DFS(G, w)$
        $if\ (killed = true)$
            return $true$
```

3. Let graph $G = (V, E)$ be an undirected graph with $k > 1$ connected components. Design an algorithm that adds $k - 1$ edges to G to make it connected and analyze its complexity. Assume that you can add an edge $(u, v) \notin E$ in constant time by invoking the method $G.addEdge(u, v)$.

Solution

```
$ConnectDFS(G, v)$
    for $v \leftarrow 1\ to\ n$: do
        $LV\ [v].ID\ \leftarrow\ 0$
    $pv \leftarrow NIL$
    for $v \leftarrow 1\ to\ n$ do
        if($LV\ [v].ID = 0$) then
            $DFS(G, v)$
        if ($pv\ != \ NIL$) then
            $G.addEdge(pv, v)$
        $pv \leftarrow v$
```

Correctness: The algorithm works in that it is an enrichment of $DFS(G, v)$ in which the variable pv indicating the starting node of the previous connected component C_{s-1}.

When I identify an unconnected component with starting node v, it is called $DFS(G, v)$: in this way I select all nodes in C_s ($\forall\ v \in C_s, LV[v].ID\ \leftarrow\ 1$).

Once $DFS(G, v)$ is finished, the algorithm checks if a connected component C_{s-1} has already been found previously, with starting node starting pv. If yes, it proceeds to connect the starting nodes of Cs and C_{s-1} by calling a function $G.addEdge(pv, v)$.

Complexity: $O(n + m)$, same as for DFS.

4. Let be the unconnected graph $G = (V, E)$ with n vertices and m edges. Design an algorithm that counts the pairs of vertices $u, v \in V$ such that u and v are reachable from each other via paths, analyzing their complexity. To have full score, the complexity must be $O(n + m)$ (let us recall that from a set of K objects one can form $\frac{K(K-1)}{2}$ distinct pairs).

<u>Solution</u>

The idea is to use a modified BFS to determine, for each connected component of G, its K cardinality by adding the value $\frac{K(K-1)}{2}$ to the count of pairs of vertices reachable one from each other.

Suppose we have modified $BFS(G, v)$ by defining a cardinality variable initialized to 0 that is incremented each time a vertex is visited and whose value is returned in the output.

Algorithm: $ReachablePairs(G)$

Input: undirected and unconnected graph $G = (V, E)$

Output: Number of pairs u, v ∈ V reachable from each other

```
$count \leftarrow 0$
    for $v \leftarrow 1\ to\ n$: do
        $if\ (LV\ [v].ID\ =\ 0)\ then$
            $K \leftarrow BFS(G, v)$
            $count \leftarrow count + \frac{K(K-1)}{2}$
    return $count$
```

Analysis

The correctness of the algorithm is immediate (given we simply use BFS and then add a constant number of objects), and the changes made to BFS do not alter its complexity, leaving it intact at $O(n + m)$.

5. Let be the directed and connected graph $G = (V, E)$ in which each vertex has degree exactly c, with $c > 2$. Let us consider the execution of $BFS(G, s)$ from an arbitrary vertex $s \in V$. Prove by induction on i that the level L_i generated by $BFS(G, s)$ contains $\leq c * (c - 1)^{i-1}$ vertices $\forall\ i \geq 0$.

Base Case: $i = 0$ contains only 1 vertex.

Rec. step: Let us fix i ≥ 1 and assume, as an inductive assumption, that $|L_J| \leq c * (c-1)^{j-1}\ \forall 0 \leq j \leq i$ vertices of level L_{i+1} are all adjacent to vertices of level i, and since each vertex $v \in L_i$ has c neighbors, of which, however, at least one is in the level L_{i-1}. We conclude that:

$$|L_{i+1}| \leq (c-1) * |L_i| \leq (c-1) * c * (c-1)^{i-1} = c * (c-1)^i$$

Written by Gabriel R.

6. *The maximum spanning tree of a graph is an ST of max cost, that is, whose sum $\sum_{e \in T} w(e)$ is max. Give an algorithm for this problem that uses an algorithm as a procedure to solve the MST problem.*

Solution

Algorithm:

- multiplies the weights of all sides by -1.
- applies Kruskal's algorithm

7. *Consider a directed graph $G = (V, E)$ with weights on nonnegative sides. Under what conditions is there a unique shortest path from $s \in V$ to $t \in V$?*

- *When all weights are positive and distinct integers*
- *When all weights are powers of 2 distinct*
- *When it is worth 1 and the graph contains no direct cycles*

Solution

The second one is the correct answer: Two sums of distinct powers of 2 can never be the same number (consider that the numbers are written in binary). For 1 and 3 there are counterexamples.

8. *Consider a directed graph $G = (V, E)$ with weights on nonnegative sides. Let the bottleneck of path as the maximum weight of its sides (instead of the sum of the weights on all sides). Modify Dijkstra's algorithm to compute, $\forall\ v \in V$, the smallest bottleneck on all possible paths from a starting node s to v. The algorithm must have complexity $O(m * n)$.*

Solution

Replace, in the 1st in the 3rd line of the while loop of Dijkstra's algorithm,$len(v) + l_{(v^*,w^*)}$ with $\max\{len(v), l_{(v,w)}\} + l_{v^*,w^*}$ with $\max\{len(v^*\}, l_{(v^*,w^*)}\}$

```
$Dijkstra(G, s)$
    $X \leftarrow \{s\}$
    $len(s) \leftarrow 0$
    $len(v) \leftarrow +\infty\ \forall\ v \neq s$
    while $there\ is\ an\ edge\ v, w\ with\ v \in X, w \notin X$: do
        $(v^*, w^*) \leftarrow such\ an\ edge\ minimizing\ \max\{len(v), l_{(v,w)}\}$
        $X \leftarrow X \cup w^*$
        $len(w^*) = \max\{len(v^*), l_{(v^*,w^*)}\}$
```

9. *Let G^c be the complement graph of G (contains exactly those sides that are not present in G). If V^* is a vertex cover of G then it holds that $V \setminus V^*$ is a clique of maximum size in G^c(clique: the subgraph largest complete there is). Is it possible to approximate the clique problem by applying a 2-approximation algorithm for vertex cover?*

Solution

No, in general the reduction functions between problem don't preserve their approximation factor. In this case, the size of the clique of maximum size in G is $|c_{max}| = \frac{n}{2}$. So in G^c: $|V^*| = n - \frac{n}{2} = \frac{n}{2}$. We apply $Approx\_Vertex\_Cover(G^c) \rightarrow V'$. It could be $|V'| = n$ so $2|V^*|$. So, the clique of size $n - n = 0$ is returned and does not approximate for any $\rho(n)$.

10. *The problem is: Given $G = (V, E)$ find a loop if it exists. Design and analyse an algorithm based on BFS to solve the problem with complexity $O(n + m)$. Hint: use the fact that there exists a loop in G if and only if the execution of BFS labels one edge as CROSS EDGE.*

Solution

Without loss of generality, assume the graph is connected (if not, do the following for each connected component). Pick an arbitrary vertex as root and do a BFS, while maintaining a BFS tree.

The BFS tree initially contains the root and at each point, when you encounter an edge to a new vertex, the new vertex is added to the tree and the edge is added between the new vertex and the parent. If at any point you encounter an edge from a node to a seen vertex, then there is a cycle in the graph, and you can return the cycle by tracing the path from each end point of the latest edge backwards in the BFS tree till they meet (linear time) and displaying it in the correct order. If the BFS ends without incident, there is no cycle. BFS runs in linear time on the edges/vertices, as each edge/vertex is visited only once.

```
procedure $BFS(G, s)$
    $visit(s)$
    $L_V[s].ID = 1$
    $Create\ a\ set\ L_0\ containing\ s$
    $i = 0$
    while $(!L_i.isEmpty())$ do:
        $Create\ a\ empty\ set\ of vertices\ L_{i+1}$
        for each $v \in L_i$ do:
            for each $e \in G.incidentEdges(v)$ do:
            if $L_E[e].label = null$ then
                $w = G.opposite(v, e)$
                if $L_v[w].ID = 0$ then
                    $L_E[e].label = DISCOVERY\ EDGE$
                    $visit\ w$
                    $L_V[w].ID = 1$
                    $L_V[v].parent = u$ // $Set\ the\ parent\ of\ v\ as\ u$
                    $add\ w\ in\ L_{i+1}$
                else
                    $L_E[e].label = CROSS\ EDGE$
                    $// Found\ a\ cycle, return\ the\ cycle$
                    $return\ get\_cycle(u, v, e)$
    $i = i + 1$
$procedure\ get\_cycle(u, v, e)$:
    $// Trace\ the\ path\ from\ u\ to\ v\ in\ the\ BFS\ tree$
    $path_u = []$
    $curr = u$
    $while\ curr\ != v$:
      $path_u.append(curr)$
      $curr = L_V[curr].parent$
    $path_u.append(v)$
    $path_u.reverse()$
    $// Trace\ the\ path\ from\ v\ to\ u\ in\ the\ BFS\ tree$
    $path_v = []$
    $curr = v$
    $while\ curr\ != u$:
      $path_v.append(curr)$
      $curr = L_V[curr].parent$
    $path\_v.append(u)$
    $// Construct\ the\ cycle\ by\ combining\ the\ two\ paths\ and\ the\ edge\ e$
    $cycle = path_u + [e] + path_v[1:]$
  $return\ cycle$
```

11. Let $G = (V, E)$ be a disconnected graph with n vertices and m edges. Design an algorithm to count the number of pairs $u, v \in V$ such that u and v are connected with a path. Analyse the complexity of the algorithm. There is a solution with complexity $O(n + m)$.

<u>Solution</u>

To count the number of pairs, we simply need to count the number of edges connected between each traversal done by the algorithm, may it be DFS or BFS.

```
$function\ countPairs(G, V)$:
  $count = 0$ $// Initialize\ count\ of\ pairs$
  $for\ each\ v\ in\ V$:
    $if\ L_V[v].ID\ =\ 0$: $//\ If\ vertex\ v\ is\ not\ visited$
      $visit(v)$      $//\ Mark\ vertex\ v\ as\ visited$
      $count\ +=\ DFS(G,v)\ -\ 1$ $//\ Increment\ count\ by\ the\ number\ of\ reachable\ vertices\ from\ v$
  $return\ count$
$procedure\ DFS(G,v)$:
  $visit(v)$
  $L_V[v].ID\ =\ 1$
  $//\ Initialize\ count\ to\ include\ the\ current\ vertex$
  $count\ =\ 1$
  $for\ all\ e\ in\ G.incidentEdges(v)$:
    $if\ L_E[e].label\ =\ null$:
      $w\ =\ G.opposite(v,e)$
      $if\ L_V[w].ID\ =\ 0$:
        $L_E[e].label\ =\ DISCOVERY\ EDGE$
        $//\ Recursive\ call\ to\ DFS$
        $count\ +=\ DFS(G,w)$
    $else$:
      $L_E[e].label\ =\ BACK\ EDGE$
  $return\ count$
```

12. Let $G = (V, E)$ be a graph with n vertices and m edges. Develop an algorithm that returns a vertex $i \in V$ that can reach (with paths) $\geq \frac{n}{2}$ other vertices. Analyse the complexity of the algorithm. If such a vertex does not exist the algorithm returns null.

Solution

Here's the algorithm:

1. Initialize an empty set $reachable$ to store the vertices reachable from each vertex.
2. For each vertex v in V, perform a DFS traversal starting from vv to mark all reachable vertices.
3. During each DFS traversal, keep track of the vertices reachable from v and add them to the set.
4. After completing DFS traversal for all vertices, iterate through each vertex ii and check if the number of vertices reachable from i is greater than or equal to $\frac{n}{2}$.
5. If such a vertex i is found, return i; otherwise, return null.

```
$function\ findVertexWithReachability(G, V)$:
  $//\ Initialize\ a\ variable\ to\ store\ the\ minimum\ reachability\ count$
  $minReachability\ = \frac{|V|}{2}$
  $//\ Iterate\ through\ each\ vertex\ v\ in\ V$
  $for\ each\ v\ in\ V$:
    $//\ Initialize\ a\ variable\ to\ count\ the\ reachable\ vertices\ from\ v$
    $reachableCount\ =\ 0$
    $//\ Perform\ DFS\ traversal\ from\ vertex\ v$
    $DFS(G, v)$
    $//\ Check\ the\ reachability\ of\ each\ vertex\ w$
    $for\ each\ w\ in\ V$:
      $//\ If\ w\ is\ reachable\ from\ v, increment\ the\ reachable\ count$
      $if\ L\_V[w].ID\ =\ 1$:
        $reachableCount\ =\ reachableCount\ +\ 1$
    $//\ If\ the\ reachable\ count\ is\ greater\ than\ or\ equal\ to\ minReachability, return\ v$
    $if\ reachableCount\ >=\ minReachability$:
      $return\ v$
  $//\ If\ no\ such\ vertex\ is\ found, return\ null$
  $return\ null$
$procedure\ DFS(G, v)$:
  $//\ Mark\ vertex\ v\ as\ visited$
  $visit(v)$
  $//\ Iterate\ through\ all\ incident\ edges\ of\ vertex\ v$
  $for\ all\ e\ in\ G.incidentEdges(v)$:
    $//\ Get\ the\ opposite\ vertex\ w\ of\ edge\ e$
    $w\ =\ G.opposite(v, e)$
    $//\ If\ vertex\ w\ has\ not\ been\ visited\ yet, recursively\ call\ DFS$
    $if\ L_V[w].ID\ =\ 0$:
      $DFS(G, w)$
```

13. Let $G = (V, E)$ be the (non-direct) Facebook graph in which the vertices represents profiles, and the edges represents friendships. Assume G is connected (actually it is not). Define the separation between two profiles $u \neq v \in V$ as the number of edges in the shortest path from u to v in G. Design and analyze an algorithm to determine the maximum separation between two profiles in G.

Solution

Here's the algorithm:

1. *Initialize a variable $maxSeparation$ to store the maximum separation between two profiles.*
2. *For each vertex u in V, perform BFS traversal starting from u to find the shortest path to all other vertices.*
3. *During BFS traversal, maintain a distance array $dist[v]$ for each vertex v, where $dist[v]$ represents the shortest distance from u to v.*
4. *After completing BFS traversal for all vertices, iterate through each pair of profiles (u, v) and update the maximum separation if necessary.*
5. *Return the maximum separation.*

14. Can you come up with an example of a graph for which $Approx\_Vertex\_Cover$ always gives a suboptimal solution?

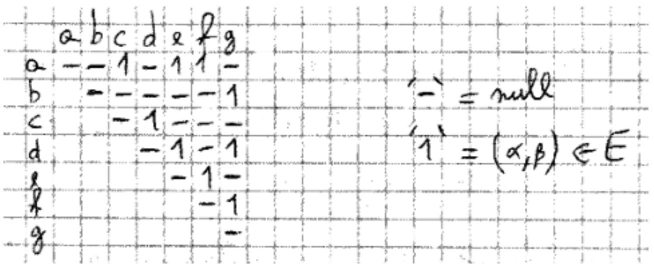
Solution

An example of a graph for which the $Approx\_Vertex\_Cover$ algorithm always yields a suboptimal solution is the star graph.

A star graph consists of a central node (or vertex) connected to $n - 1$ leaf nodes (where n is the total number of nodes in the graph). In this case, the optimal vertex cover consists of just the central node, as it covers all the edges connected to the leaf nodes.

However, the $Approx\_Vertex\_Cover$ algorithm works by iteratively selecting an edge, adding both its endpoints to the vertex cover, and removing all the edges incident to these two vertices. In a star graph, the algorithm would end up selecting all the leaf nodes along with the central node, yielding a suboptimal solution, as the optimal solution would only include the central node.

15. Given the following graph represented by the adjacency matrix:



	a	b	c	d	e	f	g
a	–	–	1	–	1	1	–
b		–	–	–	–	–	1
c			–	1	–	–	–
d				–	1	–	1
e					–	1	–
f						–	1
g							–

'–' = null

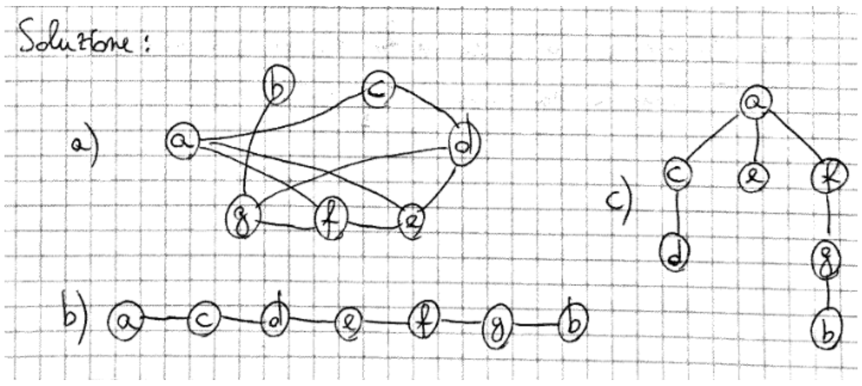
'1' = $(\alpha, \beta) \in E$

a) Draw the graph

b) Execute DFS from vertex a and show the obtained DFS tree

c) Same as previous point but for BFS

<u>Solution</u>



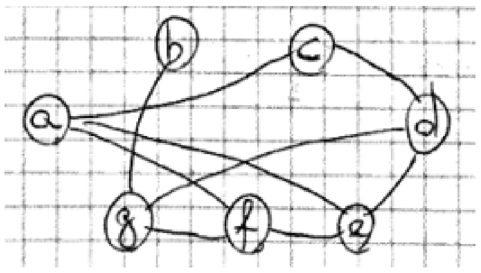
16. Given the following weighted graph, represented by an adjacency matrix:

a) List the MST edges in the order they were determined by Kruskal's algorithm

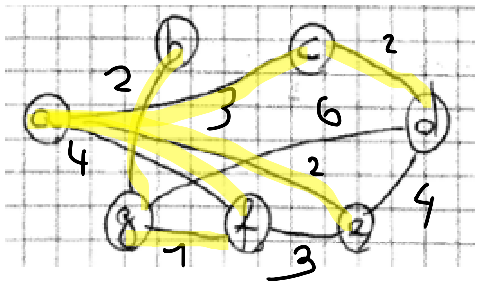
b) Do the same using Prim's algorithm

<u>Solution</u>

Given:



We should consider for example weights on this graph in order to make it work, for example:

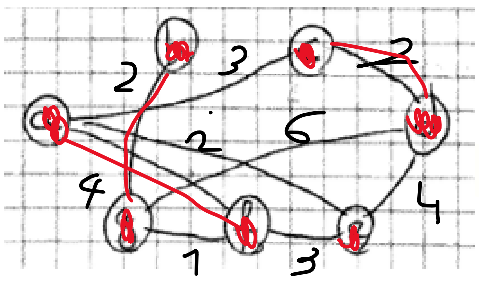


Kruskal: $\{g, f\}, \{a, e\}, \{a, c\}, \{b, g\}, \{c, d\}, \{a, c\}, \{a, f\}$

Prim:$\{a, c\}, \{a, f\}, \{a, e\}, \{a, f\}, \{b, g\}, \{c, d\}$

17. *Given the following graph, show the set of edges returned by* $Approx\_Vertex\_Cover(G)$*, briefly describing the algorithm*.

Assuming once again we may be talking here about:



Written by Gabriel R.

Remember logic and pseudocode of the approx algo:

- Choose *any* edge
- Add its endpoints to the solution
- “Remove” the covered edges
- Repeat

```
procedure $Approx\_Vertex\_Cover(G)$
    $V' = \emptyset$
    $E' = E$
    while $E' \neq \emptyset$: do
        $Let\ (u, v)\ be\ an\ arbitrary\ edge\ of\ E'$
        $V' = V' \cup \{u, v\}$
        $E' = E' \setminus \{(u, z), (v, w)\}$
        $//\ remove\ edges\ that\ have\ u\ and\ v\ as\ endpoints$
    return $V'$
```

Complexity: $O(n + m)$

So, here we would have the part in red above.

6)

(a) *Give matching and maximal matching definition*

(b) *Show a graph in which $Approx\_Vertex\_Cover$ returns a solution of cost exactly twice the optimal vertex cover*

A matching in a graph is a set of edges without common vertices, while a maximal matching is a matching which cannot be increased.

For point (b), this was shown here.