ADVANCED ALGORITHMS: EXERCISES SWISS KNIFE

Written by Gabriel R.

1 Table of Contents

Disclaimer

This file gathers all the exercises did in class as of 2023-2024 course program. Hope this can be useful to immediately see the exercises, the categories and can be used as a file to study for the exam directly.

2 GRAPHS

2.1 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

1) In the summation, every edge is counted exactly twice

2) In a simple graph, there are $\binom{n}{2}$ possible pairs of vertices

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)

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

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 $n - 1$

2.2 DFS 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)
    - $\forall v \in V$ add a field $L_V[v].parent$
    - Modify $DFS(G, v)$ s.t. when a $DISCOVERY\ EDGE\ (v, w)$ is labeled
        - then $L_V[w].parent = v$
    - 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

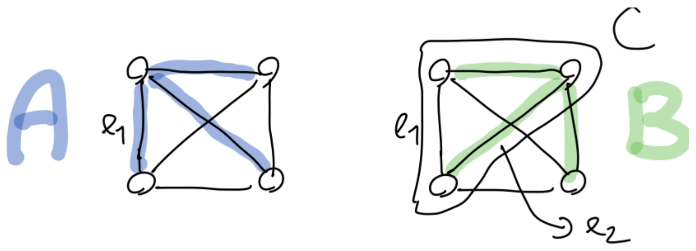
2.3 UNIQUENESS OF MSTS

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 ⇒ no cycles ⇒ C has an edge $e_2 \notin A$ $\Rightarrow w(e_2) > w(e_1)$
  - 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
- 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
- conversely, *if weights are not all distinct, generally multiple MSTs can exist*

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.

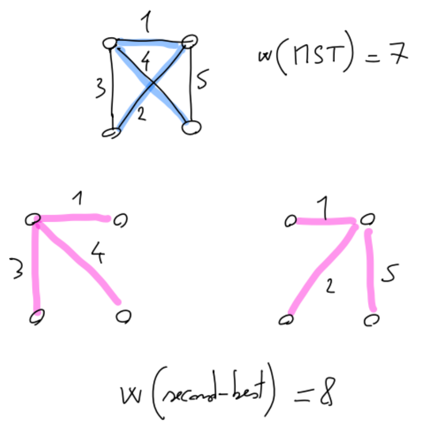
In conclusion, we might say:

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

Solution

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.



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

2.4 KRUSKAL UNION-FIND

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.

- 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

So, we have two different algorithms with complexity $O(m \log(n))$. $O(m)$ is still an open problem.

2.5 DIJKSTRA WITH HEAPS

Write an implementation of Dijkstra's algorithm with heaps.

Solution

```
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
m           $key(v) = \infty$
  ( 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 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)

2.6 Eulerian Circuit in Linear Time

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

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

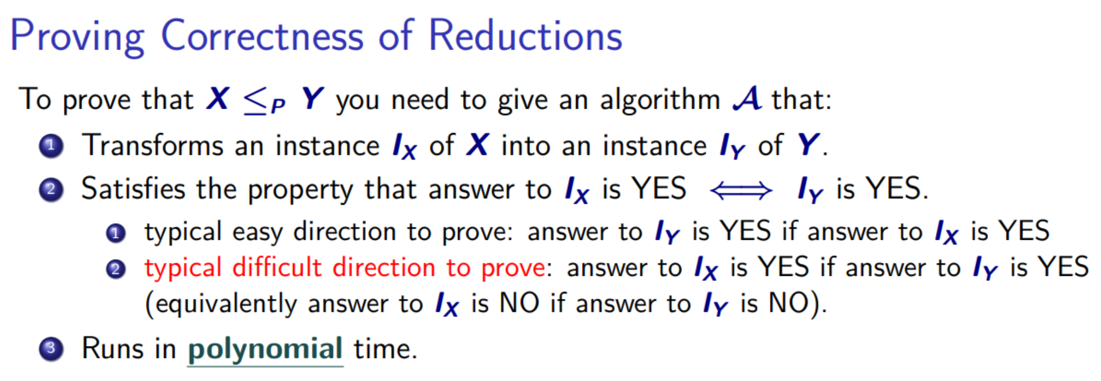
- If the partial circuit covers all edges, the algorithm is complete. Otherwise, select any vertex in the current circuit that has unused edges and start a new circuit from that vertex, merging it into the previous circuit

- 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.

3 NP-Hardness



Proving Correctness of Reductions

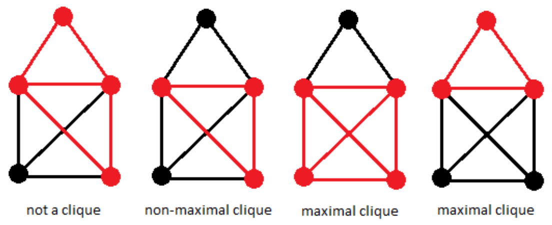
To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ that:

1. Transforms an instance I_X of X into an instance I_Y of Y.
2. Satisfies the property that answer to I_X is YES $\iff$ I_Y is YES.
   1. typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES
   2. typical difficult direction to prove: answer to I_X is YES if answer to I_Y is YES (equivalently answer to I_X is NO if answer to I_Y is NO).
3. Runs in **polynomial** time.

3.1 Clique is NP-Hard

- Maximum) Clique: compute the longest complete subgraph
  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

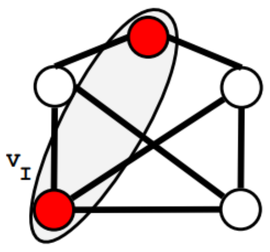
Show that Clique is NP-Hard.



Solution (a nice graphical explanation here)

Decision version:

- Input: $< G = (V, E), k >$
- Output: $\exists$ in G a clique of size k?



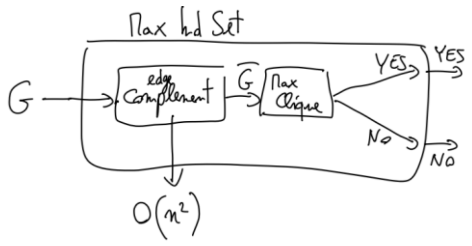
We operate a reduction from Maximum Independent Set (Ham. circuit is not really related to it; as you can see here, one can use 3SAT in order to show Clique is NP-complete). Figure here shows Independent Set.

- *Intuition*
  a. clique: vertices with all edges between them
  b. maximum independent set: vertices with no edges between them

- *Definition*
  a. given a graph $G = (V, E)$, its edge-complement $\overline{G} = (V, \overline{E})$ 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, givemn the constant work needed to traverse all edges *and* vertices:



3.2 Vertex Cover is NP-Hard

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

Show that Vertex Cover is NP-Hard.

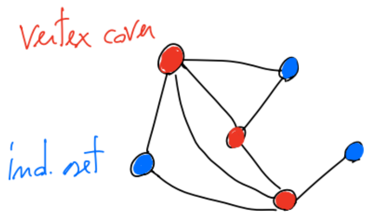
Solution (once again, a nice graphical explanation of this one here)

Decision version:

- Input: $< G = (V, E), k >$
- 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)

- *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

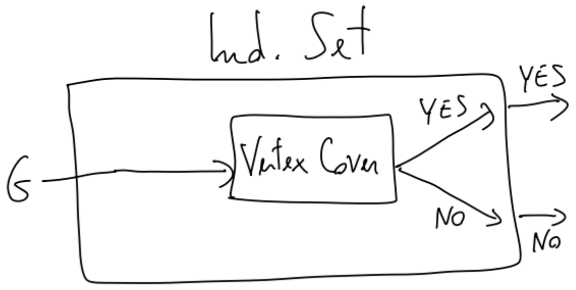


$\Rightarrow$ the longest independent set in G has size $n - k$, where k is the size of the smallest vertex cover of G

Independent set:

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

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



3.3 More Reductions can be made

Exercise

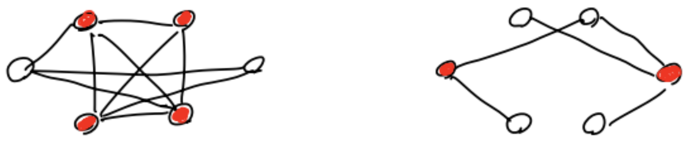
- *Show that*:
  a. $Vertex\ Cover \leq_p Independent\ Set$
  b. $Clique \leq_p Vertex\ Cover$

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



Solution (longer and better 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:

Written by Gabriel R.

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, $|V \setminus S| \geq |V| - k' = k$.

4 APPROXIMATION ALGORITHMS

4.1 LOWER BOUND FOR VERTEX COVER GREEDY ALGORITHM

Very first algorithm you can think of? Use a *greedy approach*:

- select the vertex for the highest degree
- "remove" the touched edges
- repeat

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

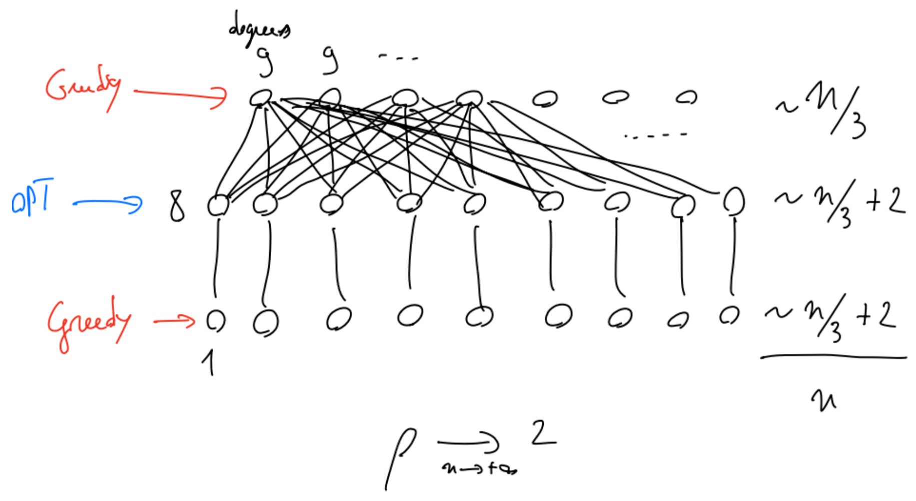
(Hint: try to prove the best you can – it should be a constant factor)

Solution

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:



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 that are all connected to the first vertex
- At the next level, add more vertices 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.

Written by Gabriel R.

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).

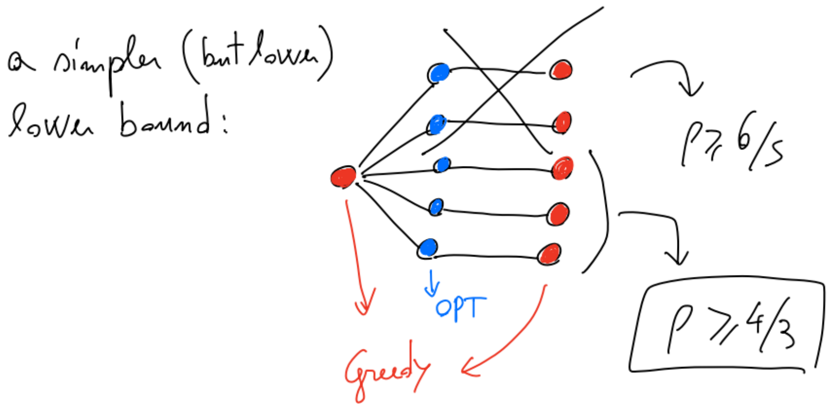
Therefore, the total size of the vertex cover produced by the greedy algorithm is: $1 + \left(\frac{n}{3} + 2\right) + \left(\frac{n}{3} + 2\right) = \frac{2n}{3} + 5$

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.

4.2 APPROXIMATION FACTOR OF APPROX VERTEX COVER

The algorithm is:

```
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: $O(n + m)$

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.

Written by Gabriel R.

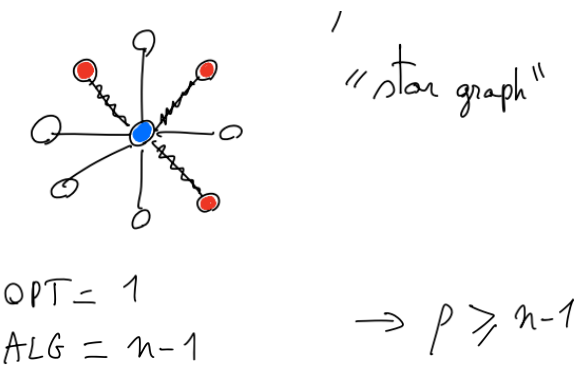
Consequently, we can establish the following inequality:

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

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

4.3 Approx Vertex Cover edit to select only one vertex

<u>Solution</u>



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.

4.4 APPROX METRIC TSP RETURNING A SOLUTION OF A SPECIFIED COST

Given the algorithm:

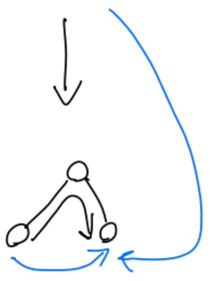
```
procedure $Approx - Metric - TSP(G)$:
    $V = \{v_1, v_2, \dots v_n\}$
    $r = v_1$ $//root\ from\ which\ Prim\ is\ run$
    $T^* = Prim(G, r)$
    $< v_{i_1}, v_{i_2}, \dots v_{i_n} \geq H' = PREORDER(T^*, r)$
    $//\ lists\ all\ the\ vertices\ in\ the\ tree\ in\ an\ ordered\ fashion\ following\ a\ preorder\ walk$
    return $< H', v_{i_1} \geq H$ $//\ basically, close\ the\ cycle$
```

Given this analysis:

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

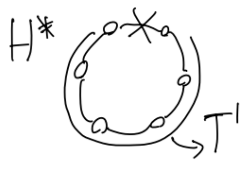


To be more precise, shortcuts in graph theory refer to edges that directly connect two vertices that are not adjacent in the original graph.

1) Lower bound to the cost of H^* (=optimal tour) (for vertex cover: $|v^*| \geq |A|$)

The weight should be: $w(H^*) \geq ?$

weights on ≥ 0 $w(T^*)$



T' is a spanning tree

⇓

$w(T') \geq w(T^*)$

This is because we obtain a spanning tree by deleting any edge from a tour, and each edge cost is non-negative.

2) Upper bound to the cut of H (the returned solution. We want to prove the following:

$$\underline{w(H) \leq \rho \cancel{w(T^*)}} \leq \rho w(H^*)$$

comes from

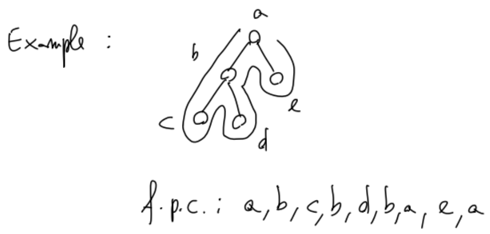
$$w(H^*) \geq w(T^*)$$

The approximation factor keeps being at most twice, so $\rho = 2$:

$$w(H) \leq 2w(T^*)?$$

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 guess (f.p.c. = "full preorder chain" from now on):



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 last occurrence of the root)
- This is like adding a *shortcut* between vertices that *does not increase the cost*

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

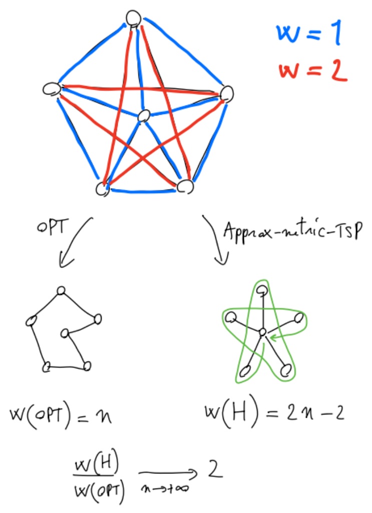
Consider a complete graph of 6 vertices. We take the edges of weight 1 (blue) and the edges of weight 2. This satisfies the triangle inequality.

Here, OPT will use only edges of weight 1, as you can see from left graph, which has weight n.

$Approx\_Metric\_TSP$ finds the minimum MST, but there is more than one to consider. Here, the star graph will be represented, with the tours of vertices, finding the shortcuts using all the red vertices.

It's basically a choice of weight 2 over all vertices not considering the central vertex, so $2 * (n-1) = 2n - 2$.

Over infinity, we have $\frac{2n-2}{n} = lim_{n \to \infty} \frac{2(n-1)}{n} = 2$



Written by Gabriel R.

4.5 SHOW THE ANALYSIS OF SET COVER IS TIGHT

Given the following analysis:

We'll show that $\frac{|F'|}{|F^*|} \leq \lceil \log_2(n) \rceil + 1$, where $n = |X|$.
OPT $\leftarrow$

Property: if (X, F) admits a cover with $|F| \leq k$, then $\forall X' \subseteq X$ (X', F) admits a cover with $|F| \leq k$.

Idea: try to bound the number of iterations such that the set of remaining elements 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 $|U_{k-i}| \leq \frac{n}{2^i}$ (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 $|F'| + +$

$\Rightarrow |F'| \leq \lceil \log_2(n) \rceil * k + 1$

$\Rightarrow |F'| \leq \lceil \log_2(n) \rceil * |F^*| + 1$ (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:

$U_k \subseteq X \Rightarrow U_k$ admits a cover size $\leq k$ all in F (i.e. not yet selected by the algorithm)
✓

(trivial) 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 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 $\cup T_i$ 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. $|U_k \cap \overline{T}| \geq \frac{|U_k|}{k}$

We'll now see that in the first k iterations, $\forall$ iteration at least $\frac{|U_k|}{k}$ 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:

$$\underbrace{|U_k|}_{\text{residual}} \le \underbrace{n - |U_k|}_{\text{covered}}$$

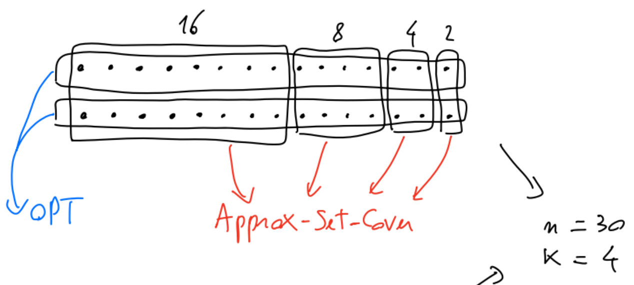
satisfied for $|U_k| \le \frac{n}{2}$ (after the k iterations, the residual universe is at least halved).

<u>Exercise</u>: *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)*

<u>Solution</u>

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

4.6 Markov's Lemma Application

Exercise

Assume that:

1) *A_Π LAS VEGAS, with $T_{A_\Pi}(n) = O(f(n))$ w.h.p; in particular, $\Pr\left(T_{A_\Pi}(n) > c * f(n)\right) \leq \frac{1}{n^d}$*
2) *A_Π has a worst-case deterministic complexity $O(n^a), a \leq d\ \forall n$*

Show that $E\left[T_{A_\Pi}(n)\right] = O(f(n))$.

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. $0 \leq t \leq b$,

$$t * \Pr(T \geq t) \leq E[T] \leq t + (b - t)\Pr\,(T \geq t)$$

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:

$$E\left[T_{A_\Pi}(n)\right] \leq \underbrace{c * f(n)}_{t} + (\underbrace{n^a - c * f(n)}_{b-t})/\underbrace{n^d}_{\Pr(T \geq t)}$$

$$\leq c * f(n) + \frac{n^a}{n^d} \leq c * f(n) + 1$$

$$= O\big(f(n)\big)$$

4.7 Show Karger is tight

Exercise

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.

Solution

Let $C_1, C_2, \dots, C_j$ denote the min-cuts of a graph $j \leq ?$

We have shown that $FULL\_CONTRACTION$ returns a particular 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:

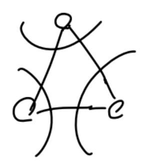
$$\Pr(A_1 \cup A_2 \cup \dots \cup A_j) = \sum_{i=1}^{j} \Pr(A_i)$$

By definition, $\Pr(A_1 \cup A_2 \cup \dots \cup A_j) \le 1$, so $\sum_{i=1}^{j} \underbrace{\Pr(A_i)}_{\geq \frac{2}{n(n-1)}} \le 1 \Rightarrow j \le \frac{n(n-1)}{2}$

This bound is tight: in a cycle of n vertices every pair of edges is a distinct min-cut:

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

$n=3$



$$\le \frac{3 \cdot \cancel{2}}{\cancel{2}} = 3$$

5 Simulation of Exam Exercises

5.1 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-approximation algorithm for Vertex Cover (Hint: show a large enough matching in the DFS tree)

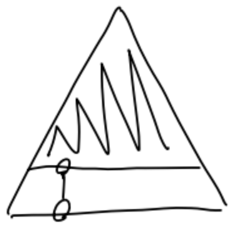
3) Show a lower bound of 2 to the approximation factor of this algorithm (it does not approximate better than 2, explained in words)

Solution

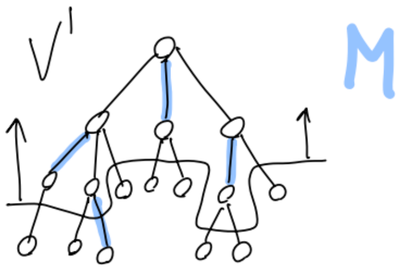
(Obviously, every vertex is covered. The question is: "Am I really covering all of the edges?". Basically, no edge can be in between of the two leaves)

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:

$$|V'| \leq 2|M|$$

Written by Gabriel R.

b) Lower bound to the cost of V^* (which is the optimal solution, selected by the algorithm: VC with min amount of vertices)

For any 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 $\forall$ edge of M.

Putting these inequalities together, we have:

$$\Rightarrow |V'| \leq \cancel{2|M|} \leq 2|V^*|$$

3) Prof. says - Do not think about complicated examples. Think about something simple.

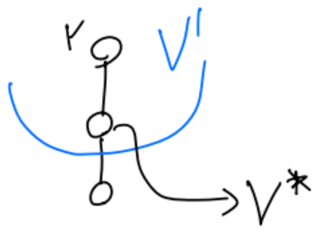
Additional question: Show that the 2-factor is tight.

Remember the following:

Specifically, if you have for instance: show the ratio is tight for a 2-approx algorithm, it means, taking for instance Vertex Cover that the ratio is exactly 2.

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

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. So, we have $\frac{|V'|}{|V^*|} = \frac{2}{1} = 2$.



A more general example is definitely the star graph, already seen at this point. So, the star graph with DFS starting from a leaf.

5.2 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 pair of points a real number that specifies their "similarity" (takes in input two points and tells how similar they are).

Smallest f = most similar points.

Goal: a k-clustering = partition of the 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 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 of (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

$\rightarrow O(n^2 \log(n))$

5.3 MAXIMUM MATCHING

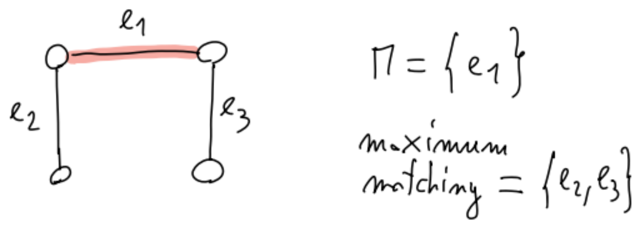
Given a graph $G = (V, E)$, recall that a matching $M \subseteq E$ is a subset of edges that do not share vertices. We want to compute a matching of maximum size (that is, containing as many edges as possible).

There exist polynomial-time algorithms, but those are slow/complicated. Consider the following simple algorithm:

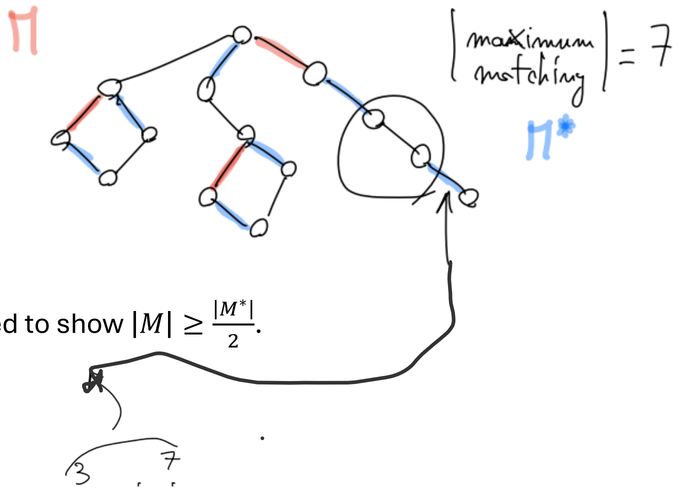
```
$GREEDY\_MATCHING(G)$
        $m = |E|$
        $M = \emptyset$
        $let\ E = \{e_1, e_2, \dots, e_m\}$
        for $i = 1\ to\ m$ do:
                if $\forall e \in M\ e \cap e_i = \emptyset$ then
                        $M = M \cup \{e_i\}$
                $return\ M$
```

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.



2) Prove that $GREEDY\_MATCHING$ is a 2-approximation algorithm (Hint: reason by contradiction)



Clearly, $|M| \leq |M^*|$, 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

⇒ ∃ edge of M^* that does <u>not</u> cover any vertex covered by edges of M

⇒ that edge(s) can be added to M, we call obtain a matching again: contradiction, given M is a maximal matching.