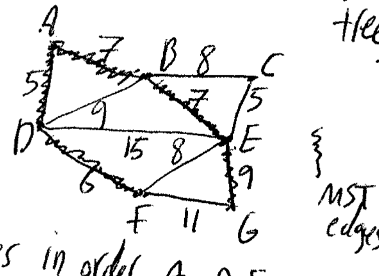
4/23

# Minimum-Cost Spanning tree (MST)

Say there is a set of computers in a office or a set of sites for VPN or a set of cities (or wiring a warehouse) that we want to connect to each other. We can model all these with an undirected graph whose edges represent connections and the weights are the lengths. We want to find a connected subgraph with minimum sum of the edge lengths. Note that the subgraph should be a tree.

The problem: Given an undirected connected weighted graph $G=(V,E)$, find a spanning tree a tree that connect every vertex, with minimum cost.



```
Algorithm prim (G); // Prim is a GREEDY algorithm
begin
    $V_{New}=\{W\}$ and $E_{New}=\{\}$, where $W$ is an arbitrary vertex
    while $V_{New} \neq V$ do
        find an edge $(u,v)$, where $u \in V_{New}$ and $v \in V-V_{new}$ with minimum weight (break tie arbitrarily)
            $V_{New}=V_{New}+\{v\}$; $E_{New}=E_{New}+\{(u,v)\}$
end;
```

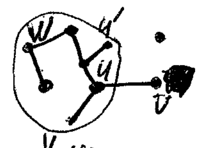
we add verties in order A, D, F, E, G

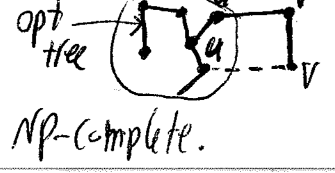
The implementation is very similar (almost identical) to Dijkstra (using heap). At each stage we can keep $SE[V]$ (instead of $SP[V]$ in Dijkstra) which keeps the minimum edge connecting $V_{New}$ to this vertex $V$ and we update it each time that we add a new vertex to $V_{New}$.
Identical to Dijkstra, using heap the running time is $O((|V|+|E|)\log|V|)$.

Why is it correct? (proof by Induction)

IH: There is alway a best solution (optimum) that has the first $k$ edges that we add to $E_{New}$.
The base is trivial since there is no edge. Note that $V_{New}$ plays the same role of $V_k$ in Dijkstra.

Pf: Consider the tree opt which has only the first $k$ edges of $E_{New}$.
We prove there is another opt' with the same cost which has the first $k+1$ edges of $E_{New}$.
Now let $(u,v)$ be the $(k+1)$th edge that we add to $E_{New}$ (and thus to the tree) when $u \in V_{New}$ and $v \notin V_{New}$. Now let add $(u,v)$ to tree opt. There should be a cycle and there is an edge $(u',v')$ where $u' \in V_{New}$ and $v' \notin V_{New}$.
Since we checked all edges going out of $V_{New}$, $w(u',v') \geq w(u,v)$.
So we can add $(u,v)$ instead of $(u',v')$, preserve connectivity, maybe lower weight and $k+1$ edges in common with new opt'. ∎
The problem is much harder if we have directed Graphs. It is NP-complete.





## NP(completeness)

Easy and Hard problems: So far we have seen many algorithms. All of them had polynomial running times, i.e., $O(n^k)$ for some constant $k$. lots of them were indeed linear $O(n)$ or quadratic $O(n^2)$. ~~But Can b~~ These problems are easy problems. But are they problems which are hard, i.e. they need exponential times such as $O(2^n)$ or $O(n!)$. The answer is that there are some wired problems for which we know, we cannot solve them in polynomial time and We need exponential time, but for ~~almost all~~ lots of normal problem, still we do not know the answer. These problems lie in the class of NP-Complete problem. Note that NP stands for Non-deteministic polynomial and not Not in Polynomial for now, but the conjecture is that indeed it is true. If you can solve this problem you will get the Millennium Prize Problem of US$1,000,000 by Clay Mathematics Institute.
lets be a bit formal:

Decision problems: are problems for which the answer is either "yes" or "no". E.g. Can we find a shortest path or an MST of cost $W$. Note that if we can solve such problems we can solve lots of optimization problems as well by a binary search.

P: is the class of all decisions problems that can be solved in polynomial time, i.e., $O(n^k)$ for some constant $k$.

EXP: is the class of all decisions problems that can be solved in exponential time, i.e. $O(2^{poly}$ where poly(n) is some polynomial in $n$, e.g. $O(2^n)$ or $O(n!) = O(n^n) = O(2^{n\log n}) \in O(2^{n^2})$

Polynomial-time verification: For many problems that maybe very hard to solve, we might hav this property that it is easy to verify whether its answer is correct.
For example consider the coloring problem: Given an undirected graph $G$, find the minimum number of colors that we can color each vertex with one of these colors and have a valid coloring, i.e., an assignment of colors to the vertices such that each vertex is assigned one color and no two adjacent vertices have the same color. 3-coloring problem asks whether we can have a valid coloring with 3 colors.
Note that though finding a 3-coloring of a graph is not easy, however if you obtain a solution with 3-colors it is easy for someone to convince that you did: Just checkin that you did not use more that 3-colors in $O(|V|)$ and checking Valid coloring in $O(|E$

Thus though we do not know any polynomial-time algorithm to decide whether the graph has a 3(valid)-coloring, there is a very efficient way to verify that a given graph has your answer as a valid 3-coloring. The solution that you provide is called a certificate. This is some piece of information which allows us to check whether the graph has a valid 3-coloring. If it is possible to verify the accuracy of a certificate for a problem in polynomial time, we say that the problem is polynomial-time verifiable.

NP: set of all decision problems that can be verified by a polynomial-time algorithm.

Note that polynomial verifiable and solving in polynomial-time are two different concepts, e.g. 3-coloring problem is NP-complete (as we see) but verifiable in polynomial time.
Also note that polynomial verification is not always easy. For example, consider the problem of determing whether a graph has exactly one valid 3-coloring. It is easy to check that there is one but not clear to show this is the only one.
Then why do we say NP(non-deterministic polynomial time) instead of VP(verifiable in polynomial time)?
Due to history, here we are referring to a non-deterministic computer which can make guesses the certificates and thus we only need to verify the certificate in polynomial time. You can learn more on this topic in other courses such as complexity theory and formal language theory.
Note that it is clear that $P \subseteq NP$, since for any problem that we can solve it in poly-time we can surely verify it in poly-time.
But still we do not know the reverse, i.e., $NP \subseteq P$ and this is the big open problem mentioned above. Many experts believe that $NP \nsubseteq P$ and thus $P \neq NP$