COMPSCI 330: Design and Analysis of Algorithms October 24, 2017

Lecture 14: Minimum Spanning Tree I

Lecturer: Rong Ge *Scribe: Fred Zhang*

1 Overview

This lecture we finish our discussion of the shortest path problem and introduce the Bellman-Ford algorithm for dealing with negative edge length. We then introduce the minimum spanning tree (MST) problem and prove a key property of MST.

2 Shortest Path with Negative Edge Length

Today we would like to deal with problem of finding shortest paths in graph when the graph has negative length edges. One of the common scenarios with negative edge is the currency exchange arbitrage.

2.1 Motivation: Currency Exchange

Suppose u, v are different currencies, and the exchange rate is $C(u, v)$ (*i.e.*, 1 unit of v is worth $C(u, v)$ units of u). We set the length of the edge as $w(u, v) = \log C(u, v)$. We take the log because, in currency exchange, the rates are multiplicative factors, but in shortest path, path length is defined as *sum* of edge lengths. By this conversion, the length of a path is log of the total exchange rate. Because of the logarithm, the edge length $w(u, v)$ can be negative or positive. It is negative if $C(u, v)$ is less than 1, and as we mentioned, a negative cycle in the graph means currency arbitrage.

In general, one can walk along a negative cycle infinitely many times, and a path that contains the cycle has length of negative infinity. Therefore, our algorithms will try to find shortest paths, assuming there is no negative cycle.

2.2 Bellman-Ford Algorithm

We have looked at the Bellman-Ford algorithm, which uses the dynamic programming paradigm. First, we define the states. Let $d[u, i]$ be the length of the shortest path to get to vertex u with (exactly) i steps. As a property of shortest path, we know that any sub-path of a shortest path is also the shortest. Therefore, the transition function is

$$d[v, i+1] = \min_{(u,v)\in E} \left(\underbrace{w[u, v]}_{\text{length of the last step}} + \underbrace{d[u, i]}_{\text{shortest path to a predecessor}} \right).$$

For this transition function, it is easy to figure out the correct ordering of evaluating the entries. We first calculate $d[v, i]$ for all v first and then are able to compute $d[v, i + 1]$. We do not have to worry about the negative cycles, as we did before.

2.3 Implementation

Before analyzing its running time, let us discuss its implementation. It follows from the guideline for implementing any dynamic programming algorithm. If you have one parameter, write one `for` loop. If you have two parameters, write a double `for` loop. Then try to fill in the table in the correct order. For Bellman-Ford, let us slightly modify the definition of state. Redefine $d[u, i]$ as the length of the shortest path to get to u with *at most* i steps. The transition function is given by

$$d[v, i+1] = \min \left\{ d[v, i], \min_{(u,v)\in E} (w\,[u+v] + d[u, i]) \right\}.$$

The difference is that we now have the option of reaching to v just using (at most) i steps when computing $d[v, i+1]$. The second part of the transition function is still the same. The implementation is fairly easy, given below by Algorithm 1.

Algorithm 1: The Bellman-Ford Algorithm

```
Initialize $d[s,0] \longleftarrow 0$, $d[u,0] \longleftarrow \infty$ for all other vertices $u$.
for $i = 1$ to $n$ do
    Initialize $d[u,i] \longleftarrow d[u,i-1]$ for all $i$.
    for all edges $(u,v)$ do
        if $w[u,v] + d[u,i-1] \le d[v,i]$ then
            $d[v,i] \longleftarrow w[u,v] + d[u,i-1]$.
if there is a vertex $u$ such that $d[u,n] \neq d[u,n-1]$ then
    Report there is a negative cycle.
```

The last part of the algorithm is to try detecting a negative cycle. Every vertex is reachable within $n-1$ steps, so if some vertex' distance to s can be decreased using n steps, compared to using $n-1$ steps, then there is a negative cycle. The proof of this fact is left as an exercise.

2.4 Example of Running Bellman-Ford

Let us now see how the Bellman-Ford algorithm runs on a graph. Consider the graph below with negative edge lengths but no negative cycle.

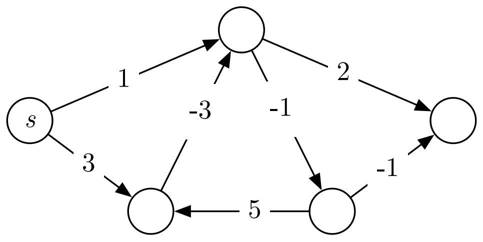
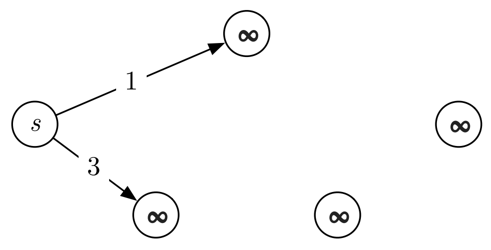
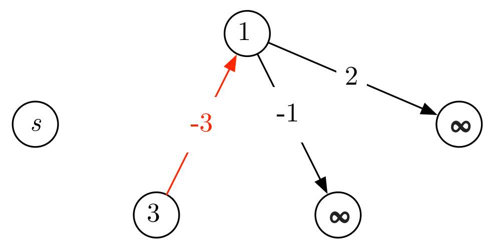


Figure 1: A graph with edge lengths and vertex labels.

Now run the Bellman-Ford algorithm starting at vertex s. For initialization, when $i = 0$, every vertex has distance $d[u, 0] = \infty$. We thus get Figure 2a.



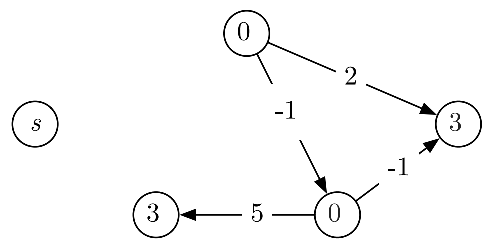
(a) The 1st iteration at $i = 0$.
Vertices are labeled with $d[u, 0]$.



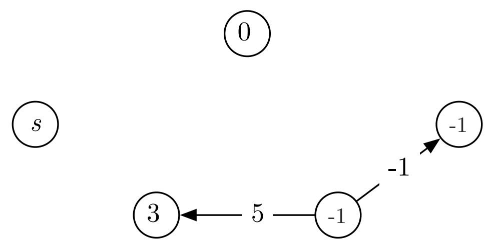
(b) The 2nd iteration at $i = 1$.
Vertices are labeled with $d[u, 1]$.

Figure 2: First two iterations.

When $i = 1$, the algorithm considers the two edges incident on s (shown in Figure 2a) and updates the two relevant vertices. Their distances become 1 and 3; see Figure 2b. Three outgoing edge from these two vertices will are relevant for the next step. The red edge is particularly important to notice. If there is no negative edge, we should not consider it at this moment, as the Dijkstra's algorithm would do. But maybe this edge is negative, and we might get a shorter path to the upper vertex by re-routing from the lower left vertex.



(a) The 3rd iteration at $i = 2$.
Vertices are labeled with $d[u, 2]$



(b) The 4th iteration at $i = 3$.
Vertices are labeled with $d[u, 3]$

Figure 3: Next two iterations.

When $i = 2$, the algorithm considers the three edges, shown in Figure 2b, and updates distances accordingly. The result is Figure 3a. Again, the algorithm will next consider all outgoing edges from vertices that were just updated. When $i = 3$, the algorithm updates the distance of two vertices and prepares to consider their outgoing edges.

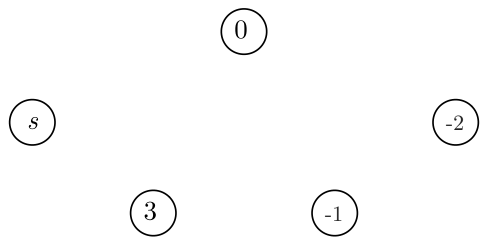


Figure 4: The 5th iteration at $i = 4$. Vertices are labeled with $d[u, 4]$, the final solution.

When $i = 4$, this is the last iteration, because $n = 5$ and we know there is no negative cycle. The algorithm makes one more update this iteration.

2.5 Analysis

Correctness. We have understood that if there is no negative cycle in the input graph, then this algorithm based on dynamic programming is correct. However, how can the algorithm correctly detect a negative cycle if there is one? We need two claims.

- When the graph has no negative cycle, a shortest path can only contain $n-1$ edges. Intuitively, this is because if there is no negative cycle, then the shortest path should not visit a vertex twice. Of course, if no vertex is repeated in a shortest path, every shortest path is contains at most $n - 1$ edges. The proof is left as an exercise.

- On the other hand, we also want to prove that if the graph has a negative cycle, then there is a vertex whose shortest path with n edges is shorter than all paths with at most $n - 1$ edges. Then in the last iteration of the algorithm ($i = n$), one vertex' distance label will decrease, indicating that there is a negative cycle.

Running Time. We have seen a naïve implementation of the Bellman-Ford algorithm (Algorithm 1). What is its running time? Each iteration takes $O(m)$ time, since the algorithm checks, for every edge, if it can be used to update a shortest path. There are n iterations, so the total running time is $O(mn)$.

This is clearly slower than the Dijkstra's algorithm, which runs in time $O(m \log n)$. The gap here is almost a factor of n. This is usually a big deal because if you have a graph of just $10,000$ vertices, then the running time would be $10,000$ slower, and it is pretty bad. In practice, people use many heuristics, though they do not affect the worst-case analysis of $O(mn)$. A major heuristic is to only consider outgoing edges from vertices that were updated last iteration. Observe that this is exactly what we have done in Section 2.4.

3 Minimum Spanning Tree

Now let us go to minimum spanning tree (MST). The algorithms for MST are fairly easy, but proving correctness is much harder, so we are going to spend more time on that. Formally, MST is the following problem.

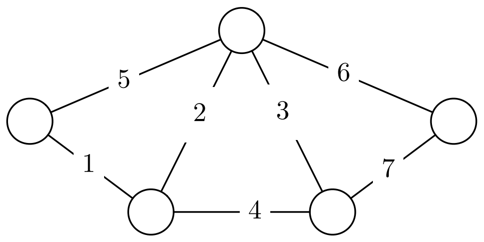
- Input: an undirected graph with edge weights. (There is another version of the problem defined on directed graph, but it requires very different algorithms.) The edge weight $w[u, v]$ is the cost of connecting two vertices u, v. We assume they are non-negative.

- Goal: select a subset of edges such that every pair of vertices can be connected by these edges. Minimize the total weight of the edges selected.

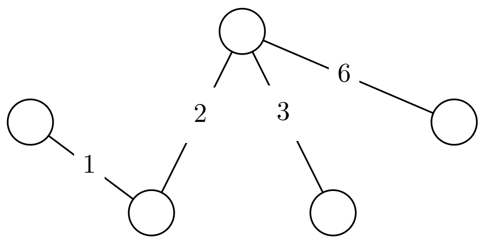
3.1 Example

Now let us take the graph below as an example (Figure 5a) and consider its MST. Our goal is just to select a set of edge connecting every vertex. Of course, if one keeps every edge, the graph is connected. But we would like to remove some to minimize the total weights of edge that are left.

The final solution (Figure 5b) doesn't look like a tree because we didn't draw it upside down and name a root. But in general, a tree is just a connected graph with no cycle. The graph in Figure 5b clearly qualifies for that.



(a) Input graph with non-negative edge weights



(b) Solution. An MST of cost 12.

Figure 5: An MST instance.

We call this a spanning tree because all edges are from the original graph. In other words, we are required to return a tree as a subgraph of the input. Moreover, to connect every vertex, we should pick a tree for minimum cost. If there is a cycle, we could remove an edge from the cycle, and the graph remains connected.

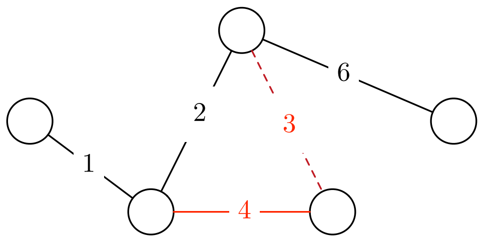
4 Key Property of MST

Previously in shortest path, we identified the shortest path property. This leads to many dynamic programming-based algorithms. In MST, we follow the same idea and look for a key property that enables algorithm design.

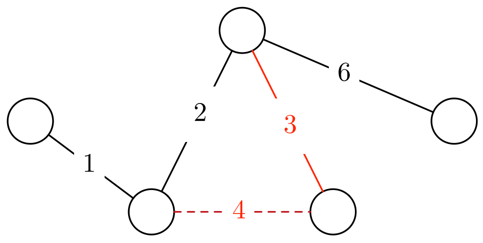
Here, one may wonder if an analogy of the shortest path property would hold. Namely, a subtree T of an MST is also an MST for the vertices in T. In fact, this is true because if there is another cheaper tree T' connecting these vertices, one should have picked T' instead of T in the large MST, as this decreases the cost. Now can we use dynamic programming based on this property? The problem is that there are 2^n subproblem, where we want to find an MST for every subset of vertices. This cannot lead to an efficient algorithm.

4.1 Swap Operation

Let us go back to MST and observe that adding any non-tree edge to a tree will introduce a cycle and, on the other hand, for a graph with a cycle, removing any edge in the cycle results in a tree. Now let us make it a basic operation of adding an edge and removing another one in the cycle created, and call this a *swap*. This operation is useful. Consider again Figure 5a.



(a) A suboptimal spanning tree of cost 13.



(b) Swapping the red edges gives MST.

Figure 6: A swap operation.

Why are we considering this operation? Because we will use it in the proof of the algorithms' correctness. It turns out the MST algorithms are greedy algorithms. Recall that for proving greedy algorithms, the general recipe is to assume optimal solution is different from the current solution. Then try to modify the optimal solution to make it look similar to the algorithm's solution. We will see soon how the swap operation is used in the proof.

4.2 Cuts in Graph

Before delving into details, let us define the notion of the *cut* of a graph. A cut is a set of edges that separates the vertices into two parts. Usually, we specify a cut by a subset of vertices $\left(S, \overline{S}\right)$, where S is a set of vertices and $\overline{S}$ is the remaining vertices.

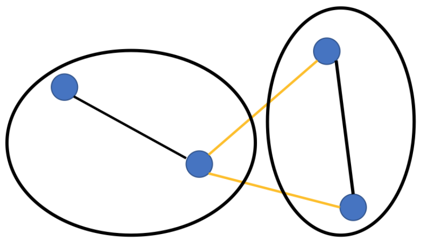


Figure 7: A graph cut. The yellow edges are cut edges.

Now we want to design a greedy algorithm. Recall that a greedy algorithm breaks down the problem into a sequence of decisions and for each decision makes the obvious choice. Usually it is easy to come up with the obvious choice, but not so clear in MST.

So we need the notion of graph cut. Notice that from every cut $\left(S, \overline{S}\right)$, an MST must select at least one edge in order to be connected. Otherwise, a vertex in S is not connected to a vertex in $\overline{S}$, so it is not a valid spanning tree. Now since we want to be greedy, let us choose the cheapest edge for each cut. This is the obvious choice.

4.3 Key Lemma

Why is this greedy rule good? We claim the following key lemma.

Lemma 1 (Key Lemma)**.** Suppose F is a set of edges inside some MST T. If $(S, \overline{S})$ is a cut that does not contain any edge in F and e is the minimum cost edge in the cut, then it is safe to add e to F.

Let us understand this claim. Suppose we already have a subtree F of some MST. Let $(S, \overline{S})$ be a cut not connected by F yet and e be the cheapest cut edge. The lemma claims that adding e to F keeps us on the right track. Formally, $F \cup \{e\}$ is still a subtree of some MST T'. Hence, if we keep adding edges in this manner, we would end up with some MST.

Proof (Key Lemma). The proof will be quite graphical. Fix a cut $(S, \overline{S})$ that F has not connected yet. Consider the MST T that contains F. In Figure 8, green edges are those in T but not in F.

Now consider we add e into T. As we can see, it will create a cycle in the middle. Since this cycle crosses the cut $(S, \overline{S})$, there must be another edge $e' \in T$ that is in the cycle and is a cut edge. We swap e and e'. The cost of e' certainly cannot be less than e, since we assume e is the cheapest

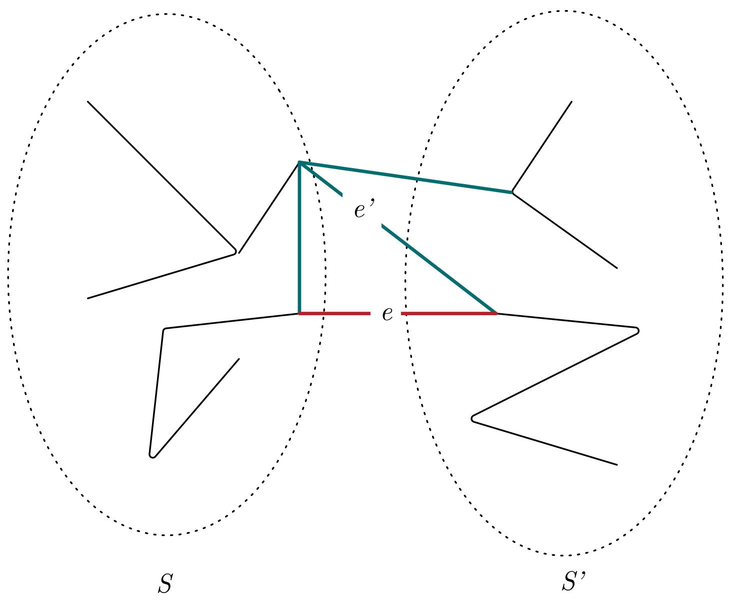


Figure 8: Swap e and e'

one in the cut. Therefore, the swap does not increase the cost. Moreover, after the switching, we still get a spanning tree. Therefore, if $T' = T \cup \{e\} \setminus \{e'\}$, $w(T') \leq w(T)$, and the MST T' contains F. This completes the proof. $\square$