CS 3510 — Algorithms, Spring 2022

Lecture 16: Bellman-Ford & Floyd-Warshall

March 17, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

1 Continued Proofs

Theorem. *Let $e^\star$ be the lightest edge in G, then every MST of G includes $e^\star$*

Proof. Assume as a contradiction that T is an MST that doesn't contain $e^\star$, then we add $e^\star$ to T.

We also know that adding $e^\star$ to T causes T to have a cycle. Since T is connected, there is a path from one endpoint of $e^\star$ to the other.

Hence, since T has a cycle, there is some vertex F that is part of the same cycle as $e^\star$. We claim that T with $e^\star$ without F is a tree. We know that $T + e^\star - F$ is connected because it was in the same cycle as $e^\star$ (removing an edge from a cycle does not disconnect a graph) and since T originally was an MST and had $|V| - 1$ edges and we added and removed one, we still have $|V| - 1$ edges. Since it is connected with $|V| - 1$ edges, it is a tree.

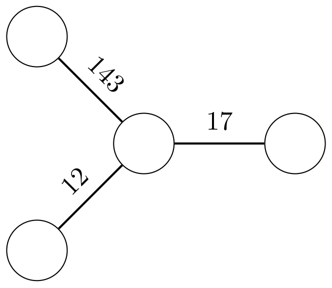
Further, we know that $T + e^\star - F$ weighs less than T because $e^\star$ was a lighter edge than F in the same cycle and F is heavier but was in the edge T. However, this is a contradiction as we assumed T (without having $e^\star$ as an edge) was an MST.

Therefore, there are no MSTs that do not contain the smallest edge $e^\star$. $\square$

1.1 Related Challenge

Prove or disprove that the heaviest edge in a graph is never in a MST.

Counterexample. See the following graph below:



In the above diagram, the 143 edge must be included in the MST as it is the only way for the top left vertex to be included.

2 Bellman-Ford

Bellman Ford is an algorithm for shortest path in G that also contains negative edges. It can also detect negative cycles (but it would not be able to find a shortest path since a graph with a negative cycle will not have a shortest path).

Consider the following function `update`:

```
1 Function update(v, w):
2     $alt \leftarrow dist(v) + weight(v, w)$
3     if $dist[w] > alt$ then
4         $dist[w] \leftarrow alt$
```

This graph algorithm has a few useful features (if initial distances are set to ∞):

1. It will never *understimate* the distance

2. If the best path from $v_0 \to w$ looks like $v_0 \to \cdots \to v \to w$ and we already know the distance to v, then `update`(v, w) will give the correct $dist[w]$.

Suppose the best path from $v_0 \to w$ looks like $v_0 \to v_1 \to \cdots \to v_i \to w$ where we start by saying $dist(v_0) = 0$ and if we ever call `update`(v_0, v_1), then $dist(v_1)$

will be correct. Afterwards, if we ever call `update`(v_1, v_2), then $dist(v_2)$ will be correct. Essentially, these calls must be made in order.

Hence, we need to call `update`(v_0, v_1), `update`$(v_1, v_2), \ldots,$ `update`(v_i, w) in that order (need not be consecutive) such that $dist(w)$ is correct.

Hence, if we call `update`(e) $\forall e \in E$, then we will at least get `update`(v_0, v_1) meaning $dist(v_1)$ will be correct. If we call it again, we will eventually call `update`(v_1, v_2) and get $dist(v_2)$ to be correct.

Hence, if we run this $i+1$ times, we will get $dist(w)$ to be correct. More generally, the maximum path length in any graph is $|V| - 1$ and therefore we can run this process $|V| - 1$ times. An implementation of this algorithm could look like:

```
Function BFord(G = (V, E), $v_0$, weights):
    $dist \leftarrow \{v : \infty \;\forall v \in V\}$
    $dist[v_0] \leftarrow 0$
    for $|V| - 1$ times do
        for $e \in E$ do
            update($e$)
```

The runtime of this is relatively straightforward as we have two nested for loops where the first for loop runs $\mathcal{O}(|V|)$ times and the second for loop run $\mathcal{O}(|E|)$ times, we get a total runtime of $\mathcal{O}(|V| \cdot |E|)$.

3 Floyd-Warshall

We are curious about all pairs of shortest paths, so we will try to build a table such that we can lookup any pair to find the shortest path starting at some vertex, ending at some vertex.

We could try to start by defining table entries of $T[u, v]$ which is the shortest path from $u \to v$. However, this does not have enough information as it is not clear what vertices are being used.Instead, we can restrict the paths to a certain length, we can then build the recurrence up from there.

Hence, we redefine the table entries as $T[u, v, k]$ which is the length of the shortest path from $u \to v$ whose intermediate vertices come from $v_1, \ldots, v_k$. For this, there are two options for $T[u, v, k]$. Either we can include v_k or we don't.

If we don't, then we can simply take $T[u, v, k-1]$ since v_k is not included. Otherwise, we can *split* the path in half since we know we will traverse through v_k. For that, we have:

$$T[u, v, k] = T[u, v_k, k-1] + T[v_k, v, k-1]$$

and all together, we are curious about the minimum math so we can take the minimum of these two cases like so:

$$T[u, v, k] = \min\left(T[u, v_k, k-1] + T[v_k, v, k-1], T[u, v, k-1]\right)$$