ICCS313 - Algorithm Analysis, T1 2019/20

Lecture 7: Minimum Spanning Trees & Shortest Paths

26 October 2019

Lecturer: Chaya Hiruncharoenvate *Scribe: -*

`* Recap Greedy Algorithm from last week`

Today we will discuss another important graph problem called Minimum Spanning Tree or MST. MST has a many applications in real-word such as networking, transportation, etc.

1 Minimum Spanning Trees

Definition 1.1. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$. H is a **spanning tree** of G if H is both acyclic and connected.

Spanning trees have many interesting properties. The following statements are equivalent:

- H is a spanning tree of G
- H is acyclic and connected
- H is connected and has $n - 1$ edges
- H is acyclic and has $n - 1$ edges
- H is minimally connected: removal of any edge disconnects it
- H is maximally acyclic: addition of any edge creates a cycle
- H has a unique simple path between every pair of nodes

Now that we have defined what a spanning tree is. Let's talk about MST.

Definition 1.2. Given a connected, undirected graph $G = (V, E)$ with edge costs c_e, a **minimum spanning tree** (V, T) is a spanning tree of G such that the sum of the edge costs in T is minimized.

[Draw a simple connected graph and identify an MST]
Instructor note: Point out that MST is not always unique.

1.1 Greedy Approach

Consider the following greedy approach: start with an empty graph and repeatedly add the next lightest edge that doesn't produce a cycle. In other words, it constructs the tree edge by edge and simply picks whichever edge is cheapest at the moment while avoid making a cycle.

This algorithm is known as **Kruskal's algorithm**. How do we know that Kruskal's tree is optimal? We will consider something called the **cut property**.

Definition 1.3. A *cut* S is a partition of nodes into two nonempty subsets: S and $V - S$

Definition 1.4. The *cutset* of a cut S is a set of edges with exactly one endpoint in S.

[Draw a graph, define a cut S and identify the cutset of S]

Proposition 1.5. *A cycle and a cutset intersect in an even number of edges*

Proof: Consider a cycle and any cut S. Idea: when there is a edge of the cycle going into S there must another edge going out of S. ∎

Lemma 1.6 (Cut property)**.** *Suppose edges X are part of a minimum spanning tree of $G = (V, E)$. Pick a cut S for which no edges of X in the cutset of S, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.*

Proof: Suppose edges X are part of an MST T. If e is a part of T, then we're done. If not, we will show that there exists another MST T' that contains $X \cup \{e\}$. Consider $T \cup \{e\}$. Since T is connected, there is already a path between the two endpoints of e. So, when e is added, there will be a cycle. Along this cycle, there must be another edge e' across the cut $(S, V - S)$. If we remove this e', we are left with $T' = T \cup \{e\} - \{e'\}$. We know that T' is a tree because T' is connected and has $n - 1$ edges. Moreover, we know that T' is a minimum spanning tree because:

$$weight(T') = weight(T) + c_e - c_{e'}$$

Since both e and e' are in the cutset of S, we know that $c_e \leq c_{e'}$. So, $weight(T') \leq weight(T)$. Since T is a minimum spanning tree, it must be the case tht $weight(T') = weight(T)$. So, T' is also a MST. ∎

In summary, the cut property says is that it is always safe to add the lightest edge across any cut (that is, between a vertex in S and one in $V - S$), provided the current partial tree X has no edges across the cut.

Corollary 1.7. *Kruskal's algorithm is optimal*

Proof: At any given moment, Kruskal's algorithm maintains a partial solution X, a collection of connected, acyclic components. The next edge e to be added connects two of these components; call them T_1 and T_2. Since e is the lightest edge that doesnt produce a cycle, it is certain to be the lightest edge between T_1 and VT_1 and therefore satisfies the cut property. So, $X \cup \{e\}$ is a part of some MST. ∎

Implmentation: $O((m + n) \log m)$ with the help of disjoint sets that allow us to do cycle checking very fast (amortized cost closes to $O(1)$).

1.2 Prim's algorithm

If we think about it, cut property tells us that any algorithm of the following form should work:

```
X = { }  // edges picked so far
repeat until |X| = |V |  1:
  pick a cut S for which X has no edges in the cutset of S
  let e in E be the minimum-weight edge between S and V  S
  X = X + {e}
```

An alternative to Kruskal's algorithm called Prim's algorithm. Here's how Prim's algorithm works. Each step the algorithm maintains a subtree X and choose S to be the vertices of X.

```
Prim(G)
```

- Choose an initial node u
- $X = \{\}$
- $cost(v) = \infty$ for all v
- $prev(v) = NULL$ for all v
- $cost(u) = 0$
- $Q = PriorityQueue(V, cost)$
- While Q is not empty,
  - $v = deleteMin(Q)$
  - For each $(v, z) \in E$:
    * update $cost(z) = min(cost(z), cost(v) + w(v, z))$
    * update $prev(z)$ as needed
    * Decrease key z in Q as needed
  - $X = X + (prev(v), v)$ (except for the u)
- return X

Running time: $O((m + n) \log n)$

2 Single-Source Shortest Paths

Given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights.

Definition 2.1. The weight $w(p)$ of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Definition 2.2. The shortest-path weight $\delta(u, v)$ from u to v is:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \to^p v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{Otherwise} \end{cases}$$

Definition 2.3. A **shortest path** from u to v is any path p such that $w(p) = \delta(u, v)$

In this lecture, we will focus on one of the many shortest-paths problem known as the **single-source shortest paths** problem. Given a graph $G = (V, E)$, we want to find a shortest path from a given source node $s \in V$ to each node $v \in V$.

If you think about it, an algorithm that solves SSSP will be able to solve the following variants as well:

- Single-destination shortest-paths problem
- Single-pair shortest-path problem
- All-pairs shortest-paths problem

3 Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. Therefore, we assume that $w(u() \geq 0$ for each edge $(u, v) \in E$.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the **minimum** shortest-path estimate, adds u to S, and relaxes all edges leaving u.

In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

`DIJKSTRA`$(G, w, s))$:

1. Initialize-Single-Soure(G,s)
2. $S = \{\}$
3. $Q = V$
4. While Q is not empty
   - $u = DELETEMIN(Q)$
   - $S = S + \{u\}$
   - For each edge $(u, v) \in E$: $RELAX(u, v, w)$

`INITIALIZE-SINGLE-SOURCE`$(G, s))$:

1. For v in V
   - $d[v] = \infty$
   - $\pi[v] = NULL$
2. $d[s] = 0$

`RELAX`$(u, v, w))$:

1. if $d[v] > d[u] + w(u, v)$ then
   - $d[v] = d[u] + w(u, v)$
   - $\pi[v] = u$

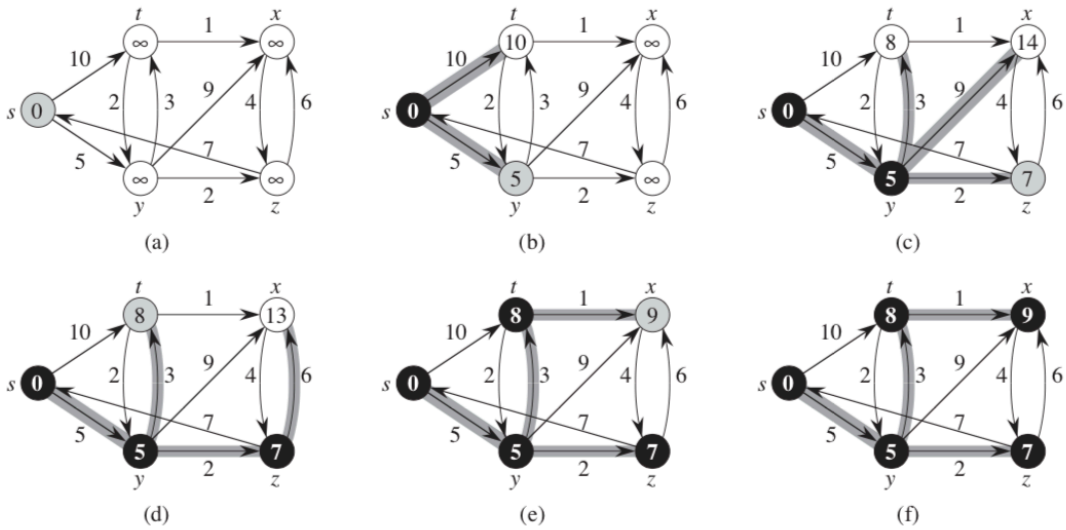
Here is example of how Dijkstra's algorithm work step-by-step:

Dijkstra's is actually using a greedy strategy to build up the solution. In each iteration, it chooses the lightest or cheapest edge to add to S. The correctness of Dijkstra's heavily relies on the nonnegative weight assumption.

Theorem 3.1. *Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source s, terminates with $d[u] = \delta(s, u)$ for all $u \in V$.*

Proof: We use the following loop invariant:

At the start of each iteration of the **while loop** $d[v] = \delta(s, v)$ for all $v \in S$



Initialization: Initially, S is empty and so the invariant is trivially true.

Maintenance: We wish to show that in each iteration $d[u] = \delta(s,u)$ for the node u added to S. For the sake of contradiction, let u be the first vertex $d[u] \neq \delta(s,u)$ when it is added to S. We know that $u \neq s$ because s is the first node added to S and that $d[s] = \delta(s,s) = 0$. Because $u \neq s$, when u is added to S, S must be non-empty. There must be some path from s to u cause $d[u] \neq \infty$. So there must be a shortest path p from s to u. Consider the first node y along the path p such that $y \in V - S$ and let $x \in S$ be a predecessor of y along p. Now, we can decompose p into $p_1 = \langle s, \ldots, x \rangle$ and $p_2 = \langle y, \ldots, u \rangle$.

We claim that $d[y] = \delta(s,y)$ when u is added to S. Since u is the first vertex that $d[u] \neq \delta(s,u)$ when added to S, we know $d[x] = \delta(s,x)$. Also, $x \in S$ and x is added to S and, the edge (x,y) is relaxed and that would set $d[y] = \delta(s,y)$.

Since $y \notin S$ and y appears before u on a shortest path p and all edges have nonnegative weights, we have $\delta(s,y) \leq \delta(s,u)$ and, thus,

$$d[y] = \delta(s,y) \leq \delta(s,u) \leq d[u]$$

Since both $u, y \notin S$ and u was chosen before y, it must be the case that

$$d[y] = \delta(s,y) = \delta(s,u) = d[u]$$

This contradicts our choice of u. Thus, we conclude that $d[u] = \delta(s,u)$ when u is added to S.

Termination: At termination, Q is empty. So $S = V$ Thus, $d[u] = \delta(s,u)$ for all $u \in V$. ∎

Running time: Running time of Dijkstra's depends on the implementation of the priority queue. For example, a binary heap implementation yeilds $O(\log n)$ for each DELETE-MIN and DECREASE-KEY. Since DECREASE-KEY will be called at most $|E|$ times and DELETE-MIN $|V|$ times, the total time is $O((|V|+|E|)\log|V|)$ or $O(|E|\log|V|)$. The running time of $O(|V|\log|V|+|E|)$ is possible with a Fibonacci heap implementation.

Final Note: Dijkstra's is very much like Prim's in that they grow the graph by adding the next lightest edge.

4 Bellman-Ford Algorithm

We have shown that Dijkstra's works on graphs containing nonnegative edges. What if a graph contains negative edges?

Dijkstra's algorithm works in part because the shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v. This no longer holds when edge lengths can be negative.

So, what do we do when a graph contains negative edges? To answer this, let's take a particular high-level view of Dijkstra's algorithm. A crucial invariant is that $d[]$ values it maintains are always either overestimates or exactly correct. They start off at ∞, and the only way they ever change is by updating along an edge:

```
def relax((u,v) in E):
    d[v] = min{d[v], d[u]+w(u,v))}
```

Here are a few observations regarding this update operation.

- This update can be applied repeatedly without hurting our shortest path.
- Dijkstra's can be thought of a sequence of updates in a particular fashion.
-
- $\underbrace{s \to u_1 \to u_2 \ldots\ldots\ldots\ldots \to t}_{P}$ Suppose P is a shortest path from s to t. It should not contain duplicate, hence the longest possible that path P can only have a maximum of $|V|-1$ edges.

So, here is an idea. Let's do update on all edges for $|V|-1$ times! So we can be sure that we cover all possibilities. This algorithm is called **Bellman-Ford** algorithm.

```
Bellman-Ford(V,E,l,s){
    For all u in V: distance(u) = infinity
    d[s] = 0
    repeate n-1 times:
        for all e in E:
            relax(e)
}
```

The running time of Bellman-Ford is $O(|V||E|)$.

5 Negative cycles

When a graph contains a negative cycle i.e. a cycle whose total weight is negative, the shortest-paths problem doesn't make much sense (because you could loop over the cycle to get a shorter and shorter path).

How do we detect if a graph contains negative cycles? This is fairly simple. Bellman-Ford can help finding negative cycles. Instead of stopping after $|V|-1$ rounds, perform an extra round of update. If some of $d[]$ reduces, then we know there must be a negative cycle.