# CS173, Minimum Spanning Trees

Tandy Warnow

December 4, 2018

## 1 Minimum Spanning Trees

A **spanning tree** of a connected graph $G = (V, E)$ is a subgraph that includes all the vertices and is a tree.

If the edges of the graph $G$ have weights, then we can also talk about the "cost" of a spanning tree $T$ for the graph: this is the sum of its edge weights. Hence, a minimum spanning tree (MST) for a graph $G = (V, E)$ is a spanning tree for $G$ that has minimum cost. This leads to the MST problem, as follows:

- Input: Connected graph $G = (V, E)$ and positive edge weights $w : E \rightarrow Z^+$
- Output: Spanning tree $T = (V, E')$ of $G$ that has minimum cost, where $cost(T) = \Sigma_{e \in E'} w(e)$

There are several well known algorithms for MST calculation, each using greedy strategies:

- Kruskal's algorithm: Keep adding the least weight edges (don't include those that create cycles)
- Prim's: Grow a spanning tree, adding least costly edge to an unvisited vertex
- Keep deleting the most costly edges, maintaining that you have a connected graph (i.e., don't delete bridges) - no name for this algorithm

All these three algorithms are polynomial time. For example, Kruskal's algorithm can be seen as having the following steps:

1. Sort the edges from lightest to heaviest
2. Initialize $T_0$ to be the empty graph (no edges) and all the vertices from $G$
3. For each edge $e$ in the list, in turn:
   - If $T_0 + e$ (the graph formed by adding $e$ to $T_0$) does not have any cycles, then replace $T_0$ by $T_0 + e$. (See below for how to test if adding an edge to a graph creates a cycle.)

When you want to find out if adding an edge $(x, y)$ to a graph $T_0$ will create a cycle, it is enough to check to see if $x$ and $y$ are in the same component of $T_0$. You can test this by starting a BFS (Breadth First Search) or Depth First Search (DFS) starting at one node (say $x$) and seeing if you reach the other node (say $y$). If you can reach $y$ from $x$ (or vice-versa), then there is a path between them in $T_0$. Therefore, adding an edge between $x$ and $y$ will create a cycle. Both BFS and DFS run in polynomial time, and are pretty common algorithms for use in graphs. Hence, Kruskal's algorithm runs in polynomial time.

**Exercises:**

- Run Kruskal's algorithm on $W_n$ (the wheel graph) where the edges that are incident with the central node have weight $n$ and the edges around the outside (i.e., the ones that are not incident with the central node) have weight 1. (For this problem assume $n \geq 3$.) What is your spanning tree?
- Run Kruskal's algorithm on $W_n$ (the wheel graph) where the edges that are incident with the central node have weight 1 and the edges around the outside (i.e., the ones that are not incident with the central node) have weight $n$. (For this problem assume $n \geq 3$.) What is your spanning tree?
- Run Kruskal's algorithm on $K_{3,5}$ with weight $w(v_i, w_j) = i + j$:
- Think about how you would implement Prim's algorithm, using DFS or BFS to check for whether you are creating a cycle when you add an edge.
- Think about how you would implement the un-named algorithm.
- Think about changing the problem so that what you want is a spanning tree that minimizes the maximum weight edge. Can you still find an optimal solution?

## 2 Triangle TSP

The Travelling Sales Person Problem (TSP) can be stated as follows. You have a set of cities and a matrix $M$ indicating how expensive it is to travel between any two cities. That cost could be miles, or tolls, or whatever - just imagine it's always positive. The TSP problem seeks a *tour* that has minimum cost. Thus, $M[i, j]$ is the cost to travel between cities $v_i$ and $v_j$.

Suppose you have an ordering $\sigma$ of the cities $v_1, v_2, \ldots, v_n$. This ordering defines a *tour* that begins at $v_1$, then visits $v_2$, then $v_3$, etc., until it reaches $v_n$, and then goes back to $v_1$. The cost of $\sigma$, denoted $cost(\sigma)$, is the sum of the distances between adjacent cities: i.e.,

$$cost(\sigma) = M[1, 2] + M[2, 3] + \ldots + M[n-1, n] + M[n, 1]$$

The TSP problem seeks the tour of minimum total cost. This is an NP-hard problem, but there are many heuristics for this problem.

One special case is where we assume that the matrix $M$ is a true "distance" matrix, which means:

- $M[x, x] = 0$ for all $x$
- $M[x, y] = M[y, x]$ for all $x, y$
- $M[x, y] \leq M[x, z] + M[z, y]$ for all $x, y, z$

The last property is called the "triangle inequality", and it may not hold on some inputs. But suppose we have a matrix where all three properties hold, so that $M$ is a distance matrix.

What we will show is that we can find an approximation algorithm for TSP when these three properties hold. We refer to this as the "Triangle TSP" problem.

# 3 Approximation Algorithms

Remember that TSP is a construction problem, where we are trying to find the minimum cost tour. Since this is an NP-hard problem, we can't expect to develop a polynomial time algorithm that always finds an optimal solution on all inputs. (This is the basic $P = NP$? question that we have talked about.)

Even though we may not be able to find a minimum cost tour, we can try to design an algorithm that produces a tour that is not *too bad*. So what do we mean by "too bad"?

For a given input matrix $M$ and a given tour $\gamma$ that we find, we refer to the "approximation ratio" as the ratio of $cost(\gamma)$ and $cost(\gamma^*)$, where $\gamma^*$ is the optimal tour for that input matrix $M$. Note that this ratio is always at least 1 on any input matrix $M$, because by definition it is not possible to get a tour that is less costly than the optimal tour.

Then, for the algorithm $A$, we define

$$r_A = \max_M \frac{cost(\gamma)}{cost(\gamma^*)},$$

where the maximum is taken over all matrices $M$. Obviously $r_A \geq 1$. What we would like is to find an algorithm where $r_A$ is as close to 1 as possible.

An algorithm $A$ that satisfies $r_A = c < \infty$ is said to be a $c$-approximation algorithm. For example, a 2-approximation algorithm for TSP is one that would always produce a tour whose cost would never be more than twice that of the optimal tour for any input.

As we will see, we can get a 2-approximation algorithm for Triangle-TSP.

# 4 2-approximation algorithm for Triangle-TSP

Here's a surprisingly simple algorithm that gives a tour that is never more than twice as "long" as the optimal tour. The input is an $n \times n$ matrix $M$, and the output will be a tour $\gamma$, for which we will prove that $cost(\gamma)$ is at most $2 \times cost(\gamma^*)$, where $\gamma^*$ is an optimal TSP.

The input is the $n \times n$ matrix $M$ that satisfies all three properties above, including the triangle inequality:

- Construct the graph $K_n$ with edge weights $w(v_i, v_j) = M[i, j]$
- Compute a MST $T_0$ on the edge-weighted graph you constructed
- Double the edges in $K_n$, creating a graph $G$
- Find an Eulerian tour for $G$, and call this $\gamma$
- Replace $\gamma$ by a tour $\gamma'$ that has each vertex appearing only once (do this by starting at any node in the tour, then listing each node only the first time it appears).

**Theorem:** The algorithm described above is a 2-approximation algorithm; thus, $cost(\gamma') \leq 2 \times cost(\gamma^*)$.

**Proof:** First, we show that $cost(T_0) < cost(\gamma^*)$. Remove any edge in $\gamma^*$; this produces a spanning tree $T$ for $G$; note that $cost(T) < cost(\gamma^*)$ since all edge weights have positive weight. Since $T_0$ is a MST, this means that $cost(T_0) \leq cost(T)$. Putting this all together, we obtain $cost(T_0) \leq cost(T) < cost(\gamma^*)$.

Now remember how we compute $\gamma$: we have doubled every edge in $T_0$ (the MST), and then computed an Eulerian tour $\gamma$. Because every edge is doubled, we get $cost(\gamma) = 2 \times cost(T_0)$. We then modified $\gamma$ to get a tour $\gamma'$ that only visits every vertex once. Because $M$ satisfies the triangle inequality, we get $cost(\gamma') \leq cost(\gamma)$. Hence, $cost(\gamma') \leq 2 \times cost(T_0)$.

Therefore, $cost(\gamma') \leq 2 \times cost(T_0) < 2 \times cost(\gamma^*)$. In other words, $\gamma'$ is a tour that is less than twice as costly as the least costly tour. In other words, for all inputs $M$, the algorithm produces a tour that is less than twice as costly as the least costly tour.