Lecture 17 — Minimum Spanning Tree

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Margaret Reid-Miller — 21 March 2013

What was covered in this lecture:

- Minimum Spanning Tree

1 Minimum Spanning Tree

The minimum (weight) spanning tree (MST) problem is given an connected undirected graph $G = (V, E)$, where each edge e has weight $w_e \geq 0$, find a spanning tree of minimum weight (i.e., the sum of the weights of the edges). That is to say, we are interested in finding the spanning tree T that minimizes

$$w(T) = \sum_{e \in E(T)} w_e.$$

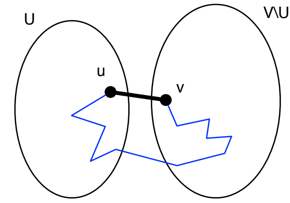
You have seen Minimum Weight Spanning Trees in 15-122 and possibly in 15-251. These classes went over Kruskal's and Prim's algorithms. At a glance, Kruskal's and Prim's seem to be two drastically different approaches to solving MST: whereas Kruskal's sorts edges by weight and considers the edges in order, using a union-find data structure to detect when two vertices are in the same component and join them if not, Prim's maintains a tree grown so far and a priority queue of edges incident on the current tree, pulling the minimum edge from it to add to the tree. The two algorithms, in fact, rely on the same underlying principle about "cuts" in a graph, which we'll discuss next.

Light Edge Rule. The main property that underlines many MST algorithms is a simple fact about cuts in a graph. Here, we will assume without any loss of generality that all edges have distinct weights. This is easy to do since we can break ties in a consistent way. For a graph $G = (V, E)$, a *cut* is defined in terms of a subset $U \subsetneq V$. This set U partitions the graph into $(U, V \setminus U)$, and we refer to the edges between the two parts as the cut edges $E(U, \overline{U})$, where as is typical in literature, we write $\overline{U} = V \setminus U$. The subset U might include a single vertex v, in which case the cut edges would be all edges incident on v. But the subset U must be a proper subset of V (i.e., $U \neq \emptyset$ and $U \neq V$).

The following theorem states that the lightest edge across a cut is in the MST of G:

Theorem 1.1. *Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights. For any nonempty $U \subsetneq V$, the minimum weight edge e between U and $V \setminus U$ is in the minimum spanning tree MST(G) of G.*

†Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.



Proof. The proof is by contradiction. Assume the minimum-weighted edge $e = (u, v)$ is not in the MST. Since the MST spans the graph, there must be some simple path P connecting u and v in the MST (i.e., consisting of just edges in the MST). The path must cross the cut between U and $V \setminus U$ at least once since u and v are on opposite sides. By attaching P to e, we form a cycle (recall that by assumption $e \notin MST$). If we remove the maximum weight edge from P and replace it with e we will still have a spanning tree, but it will be have less weight. This is a contradiction. □

Note that the last step in the proof uses the facts that (1) adding an edge to a spanning tree creates a cycle, and (2) removing any edge from this cycle creates a tree again.

As mentioned there are three algorithms for solving the MST problem that are all based on Theorem 1.1: Kruskal's algorithm, Prim's algorithm and Borůvka's algorithm. Kruskal's and Prim's are based on selecting a single lightest weight edge on each step and are hence sequential, while Borůvka's selects multiple edges and can be parallelized. We briefly review Kruskal's and Prim's algorithm and will spend most of our time on a parallel variant of Borůvka's algorithm.

Kruskal's Algorithm

The idea of Kruskal's algorithm is to sort the edges and then process them one at a time starting with the lightest edge. We maintain a set of components, where initially each vertex is its own component. When we process an edge we check if the two endpoints are in the same component. If so, we ignore the edge and move on. If not, we join the two components at the endpoints into a single component and add the edge to the MST. This joining is similar to the "edge" contraction we talked about in the last lecture, since it joins two components into one, except that we only do a single edge contraction on each step instead of multiple.

Every edge that we add to the MST is a lightest weight edge because we process the edges in sorted order starting with the minimum—i.e., it is the overall lightest weight remaining edge so it must be the lightest weight edge in the cut between its two endpoints. Efficiently joining components and checking if two vertices are in the same component can be done with a union-find data structure. With an efficient implementation of union-find, the work for the algorithm is dominated by the need to sort the edges. The algorithm therefore runs in $O(m \log n)$ work. It is fully sequential.

Prim's Algorithm

The idea of Prim's algorithm is to do a priority first search. In particular we start at an arbitrary vertex s, maintain a visited set X, and maintain priorities on the frontier vertices $v \in F$ based on the minimum weight edge from X to v. On each step we visit a vertex v with minimum priority via an edge (u, v) and add (u, v) to the MST. Since for MST we assume the graph is connected, this algorithm will visit all vertices. The algorithm is correct since the minimum weight edge leaving X must be in the MST by Theorem 1.1 (it is the minimum weigh edge separating X from $V \setminus X$). Here is the code.

```
1  function prim(G) =
2  let
3    function enqueue v (Q,(u,w)) = PQ.insert (w,(v,u)) Q

4    function prim'(X, Q, T) =
5      case PQ.deleteMin(Q) of
6          (NONE, _) ⇒ T                                  % Done
7        | (SOME(d,(u,v)),Q') ⇒
8          if (v ∈? X) then prim'(X, Q', T)              % Already visited
9          else let
10             X' = X ∪ {v}                               % Visit
11             T' = T ∪ {(u,v)}                           % Add edge to MST
12             Q'' = iter (enqueue v) Q' N_G(v)           % Enqueue v's neighbors
13         in prim'(X', Q'', T') end

14     s = an arbitrary vertex from G
15     Q = iter (enqueue s) {} N_G(s)                     % Enqueue s's neighbors
16 in
17   prim'({s}, Q, {})
18 end
```

This algorithm returns the set of edges in the MST. The only significant differences from Dijkstra's algorithm is that the weight we use for the priority queue is w instead of $w + d$. There are a few other minor changes such as maintaining the MST T and storing an edge in the priority queue instead of just the target vertex.

The algorithm runs with exactly the same work as Dijkstra's algorithm: $O(m \log n)$.

2 Parallel Minimum Spanning Tree

We now focus on developing an MST algorithm that runs efficiently in parallel. We will be looking at a parallel algorithm based on an approach by Borůvka, which quite surprisingly predates both Kruskal's and Prim's. This oldest and arguably simplest MST algorithm went back to 1926, long before computers were invented. In fact, this algorithm was rediscovered many times.

We'll use graph contraction; our presentation differs slightly from the original description. Here's the key observation:

The minimum weight edge out of every vertex of a weighted graph G belongs to its MST.

This fact follows from Theorem 1.1 by considering each vertex as our set U. We will refer to these edges as the *minimum weight edges* of the graph. The following example illustrates this situation:

where we have highlighted the minimum weight edges. Note that some edges (e.g. the ones weighted 1 and 4) are minimum for both of their endpoints.

Are the minimum weight edges all the edges of the MST? As this example shows, no—we are still missing some of the edges (in this case just the edge weighted 5). How should we proceed?

Idea #1: Throw all the minimum weight edges into the minimum spanning tree, and contract the subgraphs these edges link together. Repeat until no edges remain. In the example above, after one step we will have added the edges weighted $\{1, 2, 4, 6\}$ and be left with a graph with two vertices connected by the edge weighted 5. On the next step we add this edge and are left with a single vertex and no edges and hence we are done with the MST consisting of the edge weights $\{1, 2, 4, 6\} \cup \{5\} = \{1, 2, 4, 5, 6\}$.

This is exactly Borůvka's idea. He did not discuss implementing the contraction in parallel, but we can use the tree contraction from the last lecture notes to do this directly. Furthermore since the minimum weight edges must be a forest (convince yourself of this) we can use tree contraction which only requires $O(m)$ work and $O(\log^2 n)$ span (if using array sequences). Furthermore every step must remove at least half the vertices. Therefore the algorithm will run in at most $\lg n$ rounds and hence will take $O(m \log n)$ work and $O(\log^3 n)$ span.

Exercise 1. *Prove that each Borůvka step must remove at least 1/2 the vertices.*

Idea #2: Instead, we'll explore a slightly different idea: rather than trying to contract all these edges, we'll subselect the edges so that they are made up of disjoint stars, and we'll contract these stars using star contraction. We then repeat this simpler step until there are no edges. More precisely, for a set of minimum weight edges *minE*, let $H = (V, minE)$ be a subgraph of G. We will apply one step of star contraction algorithm on H. To do this we modify our `starContract` routine so that after flipping coins, the tails only hook across their minimum weight edge. The advantage of this second approach is that we will reduce the overall span for finding the MST from $O(\log^3 n)$ to $O(\log^2 n)$.

The modified algorithm for star contraction is as follows:

1 **function** `minStarContract`$(G = (V, E), i) =$
2 **let**
3 $\quad minE =$ `minEdges`(G)
4 $\quad P = \{u \mapsto (v, w) \in minE \mid \neg$`heads`$(u, i) \wedge$ `heads`$(v, i)\}$
5 $\quad V' = V \setminus$ `domain`(P)
6 **in** (V', P) **end**

where `minEdges`(G) finds the minimum edge out of each vertex v.

Before we go into details about how we might keep track of the MST and other information, let's try to understand what effects this change has on the number of vertices contracted away. If we have n non-isolated vertices, the following lemma shows that we're still contracting away $n/4$ vertices in expectation:

Lemma 2.1. *For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by* `minStarContract`(G, r)*. Then,* $\mathbf{E}\left[X_n\right] \geq n/4$*.*

Proof. The proof is pretty much identical to our proof for `starContract` except here we're not working with the whole edge set, only a restricted one `minE`. Let $v \in V(G)$ be a non-isolated vertex. Like before, let H_v be the event that v comes up heads, T_v that it comes up tails, and R_v that $v \in$ `domain`(P) (i.e, it is removed). Since v is a non-isolated vertex, v has neighbors—and one of them has the minimum weight, so there exists a vertex u such that $(v, u) \in$ `minE`. Then, we have that $T_v \wedge H_u$ implies R_v since if v is a tail and u is a head, then v must join u. Therefore, $\mathbf{Pr}\left[R_v\right] \geq \mathbf{Pr}\left[T_v\right]\mathbf{Pr}\left[H_u\right] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

$$\mathbf{E}\left[\sum_{v:v \text{ non-isolated}} \mathbb{I}\left\{R_v\right\}\right] = \sum_{v:v \text{ non-isolated}} \mathbf{E}\left[\mathbb{I}\left\{R_v\right\}\right] \geq n/4$$

since we have n vertices that are non-isolated. □

This means that this MST algorithm will take only $O(\log n)$ rounds, just like our other graph contraction algorithms.

Final Things. There is a little bit of trickiness since, as the graph contracts, the endpoints of each edge changes. Therefore, if we want to return the edges of the minimum spanning tree, they might not correspond to the original endpoints. To deal with this, we associate a unique label with every edge and return the tree as a set of labels (i.e. the labels of the edges in the spanning tree). We also associate the weight directly with the edge. The type of each edge is therefore (`vertex` × `vertex` × `weight` × `label`), where the two vertex endpoints can change as the graph contracts but the weight and label stays fixed. This leads to the following slightly-updated version of `minStarContract`:

```
1  function minStarContract(G = (V, E), i) =
2  let
3      minE = minEdges(G)
4      P = {(u ↦ (v, w, ℓ)) ∈ minE | ¬heads(u, i) ∧ heads(v, i)}
5      V′ = V \ domain(P)
6  in (V′, P) end
```

The function `minEdges`(G) in Line 3 finds the minimum edge out of each vertex v and maps v to the pair consisting of the neighbor along the edge and the edge label. By Theorem 1.1, since all these edges are minimum out of the vertex, they are safe to add to the MST. Line 4 then picks from these

edges the edges that go from a tail to a head, and therefore generates a mapping from tails to heads along minimum edges, creating stars. Finally, Line 5 removes all vertices that are in this mapping to star centers.

This is ready to be used in the MST code, similar to the `graphContract` code studied last time, except we return the set of labels for the MST edges instead of the remaining vertices.

```
1   function MST((V, E), T, i) =
2   if |E| = 0 then T
3   else let
4       (V′, PT) = minStarContract((V, E), i)
5       P = {u ↦ v : u ↦ (v, w, ℓ) ∈ PT} ∪ {v ↦ v : v ∈ V′}
6       T′ = {ℓ : u ↦ (v, w, ℓ) ∈ PT}
7       E′ = {(P[u], P[v], w, l) : (u, v, w, l) ∈ E | P[u] ≠ P[v]}
8   in
9     MST((V′, E′), T ∪ T′, i + 1)
10  end
```

The MST algorithm is called by running $\texttt{MST}(G, \emptyset, r)$. As an aside, we know that T is a spanning forest on the contracted nodes.

Finally we have to describe how to implement $\texttt{minEdges}(G)$, which returns for each vertex the minimum edge incident on that vertex. There are various ways to do this. One way is to make a singleton table for each edge and then merge all the tables with an appropriate function to resolve collisions. Here is code that merges edges by taking the one with lighter edge weight.

```
function joinEdges((v1, w1, l1), (v2, w2, l1)) =
  if (w1 ≤ w2) then (v1, w1, l1) else (v2, w2, l1)

function minEdges(E) =
let
   ET = {(u, v, w, l) ↦ {u ↦ (v, w, l)} : (u, v, w, l) ∈ E}
in
  reduce (merge joinEdges) {} ET
end
```

If using sequences for the edges and vertices an even simpler way is to presort the edges by decreasing weight and then use `inject`. Recall that when there are collisions at the same location `inject` will always take the last value, which will be the one with minimum weight.