Chapter 64

Sequential MST Algorithms

This chapter reviews two sequential algorithms, Prim's and Kruskal's , for computing Minimum Spanning Trees.

1 Prim's Algorithm

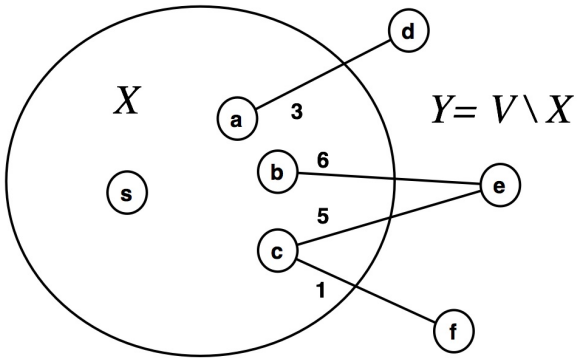
Prim's algorithm performs a priority-first search to construct the minimum spanning tree. To see the basic idea behind the algorithm, imagine that we have already visited a set X of vertices. By Lemma 63.3, we know that the minimum-weight edge e with one of its endpoint in X and the other in $V \setminus X$ is in the MST, because it is a minimum edge crossing the cut defined by X. We can therefore add e to the MST and include the other endpoint in X.

Algorithm 64.1 (Prim's Algorithm)**.** For a weighted undirected graph $G = (V, E, w)$ and a source s, Prim's algorithm is priority-first search on G starting at an arbitrary $s \in V$ with $T = \emptyset$. The algorithm visits the next vertex in the frontier with the least priority, where the priority of a vertex is defined as

$$p(v) = \min_{x \in X} w(x, v).$$

After visiting a vertex the algorithm extends the tree with the chosen edge $T = T \cup \{(u, v)\}$ when visiting v ($w(u, v) = p(v)$). When the algorithm terminates, T is the set of edges in the MST.

Example 64.1. A step of Prim's algorithm. Since the edge $(\texttt{c}, \texttt{f})$ has minimum weight across the cut (X, Y), the algorithm will "visit" `f` adding $(\texttt{c}, \texttt{f})$ to T and `f` to X.



Exercise 64.1. Prove the correctness of Prim's algorithm.

Cost of Prim's Algorithm. In the worst case, Prim's algorithm visits every edge exactly once. Because visiting an edge requires the removal of the edge from the priority queue representing the frontier, the cost is $O(\lg n)$ per edge. This is the dominating cost, and using binary heaps for the priority queue yields work and span of $O(m \lg n)$. This can be reduced to $O(m + n \lg n)$ by using Fibonacci heaps.

Prim's and Dijkstra's. Prim's algorithm is quite similar to Dijkstra's algorithm for shortest paths. Both are priority-first search algorithms. The only differences are

- Prim's algorithms starts at an arbitrary vertex instead of at a source,
- Prim's algorithm uses the priority

  $$p(v) = \min_{x \in X} w(x, v)$$

  instead of the priority used by Dijkstra's:

  $$\min_{x \in X}(d(x) + w(x, v))$$

- Prim's algorithm maintains a tree T instead of a table of distances $d(v)$.

Because of the similarity to implement Prim's algorithm, we can basically use the same priority-queue implementation as in Dijkstra's algorithm. Such an implementation requires $O(m \log n)$ work and span.

Remark. Prim's algorithm was invented in 1930 by Czech mathematician Vojtech Jarnik and later independently in 1957 by computer scientist Robert Prim. Edsger Dijkstra's rediscovered it in 1959 in the same paper he described his famous shortest path algorithm.

2 Kruskal's Algorithm

As described in Kruskal's original paper, the algorithm is:

> "Perform the following step as many times as possible: Among the edges of G not yet chosen, choose the shortest edge which does not form any loops with those edges already chosen." [Kruskal, 1956]

In more modern terminology we would replace "shortest" with "lightest" and "loops" with "cycles".

Correctness of Kruskal's Algorithm. Kruskal's algorithm is correct since it maintains the invariant on each step that the edges chosen so far are in the MST of G. This is true at the start. Now on each step, any edge that forms a cycle with the already chosen edges cannot be in the MST. This is because adding it would violate the tree property of an MST and we know, by the invariant, that all the other edges on the cycle are in the MST. Now considering the edges that do not form a cycle, the minimum weight edge must be a "light edge" since it is the least weight edge that connects the connected subgraph at either endpoint to the rest of the graph. Finally we have to argue that all the MST edges have been added. Well we considered all edges, and only tossed the ones that we could prove were not in the MST (i.e. formed cycles with MST edges).

We could finish our discussion of Kruskal's algorithm here, but a few words on how to implement the idea efficiently are warranted. In particular checking if an edge forms a cycle might be expensive if we are not careful. Indeed it was not until many years after Kruskal's original paper that an efficient approach to the algorithm was developed. Note that to check if an edge (u, v) forms a cycle, all one needs to do is test if u and v are in the same connected component as defined by the edges already chosen. One way to do this is by contracting an edge (u, v) whenever it is added—i.e., collapse the edge and the vertices u and v into a single super-vertex. However, if we implement this directly, we would need to update all the other edges incident on u and v. This can be expensive since an edge might need to be updated many times.

Union-Find. To get around these problem it is possible to update the edges lazily. What we mean by lazily is that edges incident on a contracted vertex are not updated immediately, but rather later when the edge is processed. At that point the edge needs to determine what supervertices (components) its endpoints are in. This idea can be implemented with a ***union-find data structure***. The ADT for a union-find data structure consists of the following operations on a union-find structure U:

- $insert\ U\ v$: insert the vertex v into U,
- $union\ U\ (u, v)$: join the two elements u and v into a single super-vertex,
- $find\ U\ v$: return the super-vertex in which v belongs, possibly itself,
- $equals\ u\ v$: return true if u and v are the same super-vertex. Now we can simply process the edges in increasing order.

Algorithm 64.2 (Union-Find Kruskal)**.**

```
1   kruskal (G = (V, E, w)) =
2     let
3       addEdge ((U, T), e = (u, v)) =
4         let u' = find (U, u)
5             v' = find (U, v)
6         in if (equals (u', v') then (U, T)
7            else (union (U, u', v'), T ∪ e)
8         end
9       U = iterate insert ∅ V
10      E' = sort (E, w)
11    in
12      iterate addEdge (U, ∅) E'
13    end
```

Exercise 64.2. Prove that Kruskal's algorithm correctly find the MST of a undirected graph with unique edge weights.

Cost of Kruskal's. To analyze the work and span of the algorithm we first note that there is no parallelism, so the span equals the work. To analyze the work we can partition it into the work required for sorting the edges and then the work required to iterate over the edges using union and find. The sort requires $O(m \log n)$ work. The union and find operations can be implemented in $O(\log n)$ work each requiring another $O(m \log n)$ work since they are called $O(m)$ times. The overall work is therefore $O(m \log n)$. It turns out that the union and find operations can actually be implemented with less than $O(\log n)$ amortized work, but this does not reduce the overall work since we still have to sort.