Running time of
Kruskal’s and Prim’s algorithms
for minimum spanning trees

Vassos Hadzilacos

Let $G = (V, E)$ be an undirected, connected graph, and $w(u, v)$ be an edge weight function, where $(u, v) \in E$. (We slightly abuse notation and write an edge as an ordered pair, even though the graph is undirected.) Let $n = |V|$ and $m = |E|$.

**Kruskal’s MST algorithm**

```
Kruskal(G, w)
H := array containing |E| triples of the form (u, v, w(u, v)), where (u, v) ∈ E
▷ turn H into a heap, using the third component (edge weight) as key
   BUILDHEAP(H)
▷ place each node in a set by itself
   for each u ∈ V do MAKESET(u)
   F := ∅
   while |F| ≠ n − 1 do
      (u, v, x) := EXTRACTMIN(H)
      U := FIND(u); V := FIND(v)
      if U ≠ V then
         F := F ∪ {(u, v)}
         UNION(U, V)
   return F
```

We use a heap $H$ to store the edges, using their weight as the key. We use the Union/Find data structure for maintaining disjoint sets to keep track of the connectivity of nodes via paths containing only edges in $F$. Two nodes are in the same set if and only if there is a path that connects them and contains only edges in $F$. Kruskal’s algorithm performs

- a BUILDHEAP operation on an array of $m$ entries, which takes $O(m)$ time;
- at most $m$ EXTRACTMIN operations on a heap with at most $m$ entries, each taking $O(\log m) = O(\log n^2) = O(\log n)$ time, for a total of $O(m \log n)$ time;
- $n$ MAKESET operations, each taking $O(1)$ time, for a total of $O(n)$ time;
- at most $2m$ FIND operations (two per edge), and at most $n − 1$ UNION operations (after which $|F| = n − 1$ and the while loop terminates), for a total of $O(m \log^* n)$ time.

So, the total running time of the algorithm is $O(m) + O(m \log n) + O(n) + O(m \log^* n) = O(m \log n)$. (Here we used the fact that $n = O(m)$ since the graph is connected.)
Prim’s MST algorithm

```latex
Prim(G, w)
s := an arbitrary node of G
R := \{s\}
F := \emptyset
while R \neq V do
    (u, v) := min weight edge that crosses the cut (R, \overline{R})
    F := F \cup \{(u, v)\}
return F
```

Note that the structure of this algorithm is similar to Dijkstra’s: We start at some node s; at each stage we have found a tree of minimum weight that spans a subset R of the nodes (the “explored region” of the graph), and we expand this set greedily by choosing an edge of minimum weight among the edges joining nodes in R to nodes not in R. As with Dijkstra’s algorithm, there are two implementations, one better suited for dense graphs and one better suited for sparse graphs.

In the most straightforward implementation, we maintain all edges in a list. The while loop is executed n − 1 times and in each iteration we scan the array with the edges to find a minimum weight edge that connects a node in R to a node not in R. Thus this implementation takes \(O(mn)\) time. This can be improved by maintaining an array closest, with one entry for each node, where, for each \(u \in R\), closest[u] is a node \(v \notin R\) such that \((u, v)\) has minimum weight among all edges that connect \(u\) to nodes not in \(R\). (If \(u \notin R\), we don’t care what closest[u] contains.) With this modification, finding a minimum-weight edge that crosses the cut \((R, \overline{R})\) takes \(O(n)\), instead of \(O(m)\), time. Furthermore, it is easy to update the information in closest each time a node \(v\) is added to \(R\) in \(O(n)\) time. Thus, with this implementation, Prim’s algorithm takes \(O(n^2)\) time (\(n - 1\) iterations, each taking \(O(n)\) time).

We can also use a heap to store the edges with at least one endpoint in \(R\), using the weight of each edge as the key. We can then find the minimum-weight edge connecting a node in \(R\) to a node not in \(R\) by performing repeated `ExtractMin` operations until we find an edge that crosses the \((R, \overline{R})\) cut. When a vertex \(v\) is added to \(R\), we also insert to this heap all edges \((v, u)\) for some \(u \notin R\). In this implementation, the algorithm performs \(O(m)\) `ExtractMin` and \(O(m)\) `Insert` operations on the heap. The total running time is therefore \(O(m \log m) = O(m \log n^2) = O(m \log n)\).