Topic 17, Minimum Spanning Trees: Class Solutions

Kruskal's and Prims on Unconnected Graphs

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

```
PRIM(G, w, r)
1  Q = ∅
2  for each u ∈ G.V
3      u.key = ∞
4      u.π = NIL
5      INSERT(Q, u)
6  DECREASE-KEY(Q, r, 0)        // r.key = 0
7  while Q ≠ ∅
8      u = EXTRACT-MIN(Q)
9      for each v ∈ G.Adj[u]
10         if v ∈ Q and w(u, v) < v.key
11             v.π = u
12             DECREASE-KEY(Q, v, w(u, v))
```

1. Suppose we have an <u>unconnected</u> graph, and want to find a minimum spanning <u>forest</u> (consisting of a minimum spanning tree for each connected component).

a. Will Kruskal's algorithm correctly find a minimum spanning forest in an unconnected graph? Why or why not?

> **Solution:** Yes. It considers all edges in the graph, so all edges in each connected component will be considered, thereby applying the algorithm to each component.

b. Will Prim's algorithm correctly find a minimum spanning forest in an unconnected graph? Why or why not?

> **Solution:** Yes. At first one might think that Prim's cannot reach components not connected to the start vertex *r*, but examining the operation of the loop more carefully, vertices in other components will eventually be dequeued (with key of ∞) in line 7, and construction of the MST for the given component will commence from the first such vertex dequeued. Hence, choice of r as a starting point is truly arbitrary: we could have left all keys at ∞.

Proposed MST Algorithms and their Implementations

As a working computer scientist, colleagues may propose or you may think of new algorithms to solve a problem. Critically evaluating such proposals is an important skill. Below

are three proposed MST algorithms that operate on an undirected graph G = (V, E). (One is a challenge problem.) For each algorithm,

- either prove that the set of edges *T* produced by the algorithm is always a minimum spanning tree for any G,
- or find a counter-example G where *T* is not a minimum spanning tree (either because it is not a tree, or it is not of minimum weight).
- Then, regardless of whether the algorithm is correct, identify an efficient implementation of the utility methods used. (Consider modifying existing algorithms.)

2. Maybe-MST-A

```
Maybe-MST-A (G, w)
1  T = {} // empty set
2  for each edge e ∈ E, taken in arbitrary order
3      if T ∪ {e} has no cycles
4          T = T ∪ {e}
5  return T
```

a. Does this construct MSTs? Give a proof or counter-example:

> **Solution:** Maybe-MST-A is not a valid MST algorithm. Counter-example:
> Maybe-MST-B simply constructs a tree, with no attempt to guarantee minimality. A simple counter-example is a triangle: if the two highest cost edges are chosen first, a higher cost tree will be constructed.
> Notice the similarity to the Connected-Components algorithm, which also examines edges in arbitrary order to add those that do not form cycles.

b. Describe an efficient algorithm for detecting cycles after adding an edge e, as in:

```
3      if T ∪ {e} has no cycles
```

> **Solution:** Depth-first search in O(*V*+*E*) time, exiting as soon as a back edge is found. Possible efficiency gain if you start the search with one of the vertices on the edge just added, as we know any cycle must involve this vertex, but this is still O(*V*+*E*). *Note:* this is DFS-Has-Cycle, which you wrote last week!

3. Maybe-MST-B

```
Maybe-MST-B (G, w)
1  sort the edges into nonincreasing order of edge weights w
2  T = G.E
3  for each edge e, taken in sorted order
4      if T – {e} is a connected graph
5          T = T – {e}
6  return T
```

a. Does this construct MSTs? Give a proof or counter-example:

Solution: Maybe-MST-A is a correct MST algorithm.
Note that Maybe-MST-A always maintains a connected graph, i.e. in the end, the remaining edges will constitute a spanning tree because it is connected and we have deleted the maximum number of edges we can without disconnecting it. We just need to prove that this spanning tree is minimum weight.

To prove by contradiction, suppose that when the algorithm terminates, T contains an edge e that connects two components c_1 and c_2 and could be replaced with a lower cost edge e' that must also connect these components. But then due to the ordering in line 3, e would have been processed and removed earlier than the lower weight e', because c_1 and c_2 would have remained connected by e', contradicting the assumption that the algorithm could terminate with e in T.

b. Describe an efficient algorithm for testing connectivity, as in:

```
4       if T – {e} is a connected graph
```

Solution: Depth-first search in $\Theta(V+E)$ time, checking that all vertices are reached on the first pass.

Kruskal avoids this work because it *constructs* rather than deconstructs the tree: it can then use efficient union/find, with amortized cost $O(\alpha(V))$. We can't reverse this for removing edges, as the union-find operations don't record what edge(s) led to two vertices being in the same set. This is why the constructive rather than destructive approach is preferred.