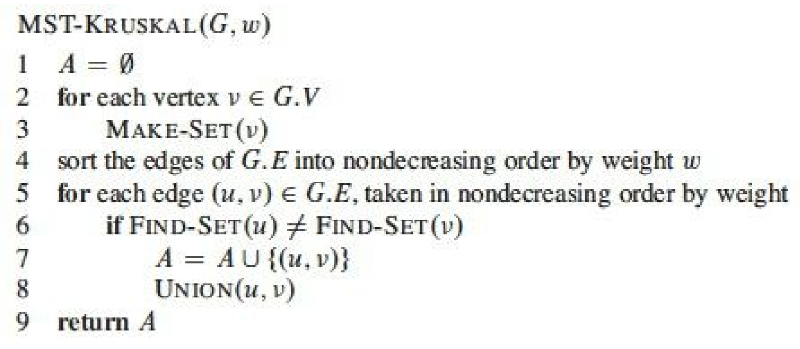
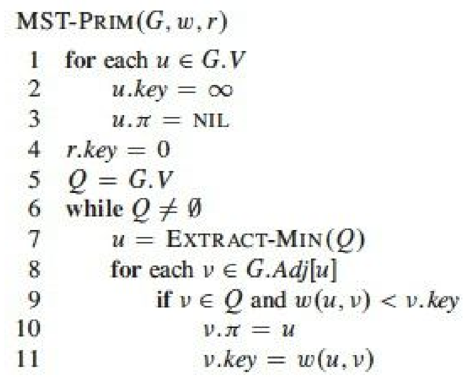
Topic 17, Minimum Spanning Trees: Class Solutions

Kruskal's and Prims



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



```
MST-PRIM($G, w, r$)
1  for each $u \in G.V$
2      $u.key = \infty$
3      $u.\pi$ = NIL
4  $r.key = 0$
5  $Q = G.V$
6  while $Q \neq \emptyset$
7      $u$ = EXTRACT-MIN($Q$)
8      for each $v \in G.Adj[u]$
9          if $v \in Q$ and $w(u, v) < v.key$
10             $v.\pi = u$
11             $v.key = w(u, v)$
```

1. Suppose we have an unconnected graph, and want to find a minimum spanning forest (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 ∞.

2. Suppose we change the representation of edges from adjacency lists to matrices.

a. What line(s) would have to change in Prim's algorithm, and how?

Solution: Line 8 would need to change as we don't have G.Adj[*u*] anymore. This would be replaced with a scan of the matrix row for *u*.
Also line 9 we would access weights looking at matrix cell A[u,v], but this is O(1) and won't affect run time.

b. What would be the resulting asymptotic runtime of Prim's algorithm, and why?

Solution: Line 8 now becomes O(V) for the row scan. (There is also a lgV operation inside the loop but O(V+lgV) simplifies to O(V).) Since this is inside the loop of line 6, which remains O(V) to extract each vertex, the overall complexity is now O(V^2).

(On the homework we may ask you to do the same thing for Kruskal.)

Is it an MST algorithm?

Below are some proposed MST algorithms that operate on a graph G = (V, E). 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).

3. Maybe-MST-A

```
Maybe-MST-A (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
```

Solution:
a. **Maybe-MST-A is a correct MST algorithm**
b. Proof:
Note that Maybe-MST-A always maintains a connected graph, i.e. in the end, the remaining edges will constitute a spanning tree. 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*.

4. Maybe-MST-B

```
Maybe-MST-B (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
```

Solution:

a. **Maybe-MST-B is <u>not</u> a valid MST algorithm**

b. 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.

5. Maybe-MST-C

```
Maybe-MST-C (G, w)
1  T = {} // empty set
2  for each edge e ∈ E, taken in arbitrary order
3      T = T ∪ {e}
4      if T has a cycle c
5          let e' be a maximum weight edge on c
6          T = T - {e'}
7  return T
```

Solution:

a. **Maybe-MST-C is a <u>correct</u> MST algorithm**

b. Proof:

The full proof is too involved to expect students to construct in class. We'll give students the half point if they make a relevant observation that can support a proof. Some key observations: we never have more than one cycle at a time in the graph (and only during lines 3-5), so it is otherwise a forest. We need to show that it is safe to remove the greatest weight edge in this cycle.

Here is a full proof:

Safe edge removal theorem: Given a graph with a single cycle, removing the edge with the maximum weight on that cycle results in a forest with the smallest total weight.

Proof: Since there is only one cycle, to create a forest, we need to remove only one edge from the cycle. Removing any edge that is not the largest from the cycle will result in a forest with a larger total weight. Q.E.D.

Claim: When Maybe-MST-C terminates, it will produce the Minimum Spanning Tree.
Proof: By induction on the number of edges inspected.
Proposition P(k): “After k edges have been inspected (after k iterations), the graph T is a minimum spanning forest (MSF) of the graph that contains only the edges inspected”.

Base case: k=1: With only one edge inspected, that edge is added, and it’s a forest and it’s of minimum weight using only the edge inspected.

P(k) => P(k+1)   (Assume P(k) is true and prove P(k+1) is true).

There are two cases to consider: (1) adding of the (k+1)-th edge does not result in a cycle, (2) adding of the (k+1)-th edge results in a cycle. Let’s consider each of these cases.

Case 1. If adding the (k+1)-th edge *e* doesn’t create a cycle, it means that *e* connects two separate subtrees. So it’s still a forest and the added edge is the only one that connects the subtrees, so the forest’s weight is minimum over the edges inspected so far.

Case 2. By inductive hypothesis, after inspecting k edges, we have a minimum spanning forest on the first k edges. Thus, If adding the (k+1)-th edge *e* creates a cycle, it is the only cycle in the forest. The safe edge removal theorem says that removing the heaviest edge on that cycle will result in a minimum forest. Thus, the resulting tree is still a MSF.

After considering the |E|-th edge, the Proposition says that T is a minimum spanning forest on the edges considered, i.e. all edges of the graph. And if G is connected, then it is a tree, and therefore, a minimum spanning tree.

(If you are spending a lot of time on this without progress, go on to the extra credit.)

Extra Credit: Efficient Implementations

Each MST algorithm above uses utility methods such as sorting, testing connectivity, detecting cycles, and finding the maximum weight edge on a cycle. The efficiency of the above algorithms depends on the efficiency of these utility methods.

6. Describe efficient implementations of each of the following utility methods. Examples of code that call the utility methods are shown below. It will be a lot easier to rely on existing algorithms that we have studied rather than invent new ones!

a. Testing connectivity, for example:
Maybe-MST-A:
```
4       if T - {e} is a connected graph
```

Solution: Depth-first search in Θ(*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(α(*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.

b. Detecting cycles after adding an edge e, for example:
Maybe-MST-B:
```
3       if T ∪ {e} has no cycles
```
Maybe-MST-C:
```
3       T = T ∪ {e}
4       if T has a cycle c
```

Solution: Depth-first search in O(*V*+*E*) time, exiting as soon as a back edge is found. Possible efficiency gain if you start 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*).

c. Finding the maximum weight edge on a cycle, for example:
Maybe-MST-C:
```
4       if T has a cycle c
5           let e' be a maximum weight edge on c
```

Solution: Assuming that we need to do this only after checking for a cycle (as is the case for Maybe-MST-C), we can modify the DFS search for detecting cycles to also keep track of the max weight edge traversed in the active recursion tree (*not* overall), returning that edge as soon as a back edge is found.