ICS 311 Fall 2020, Problem Set 09, Topics 15-17

---

1. (8 pts) Analyzing a Minimum Spanning Forest Algorithm

```
Build-MSF-By-Add-And-Fix (G, w)
1  F = {} // empty set
2  for each edge e ∈ E, taken in arbitrary order
3      F = F ∪ {e}
4      if F has a cycle c
5          let e' be a maximum weight edge on c
6          F = F − {e'}
7  return F
```

a. Complete the following proof that this algorithm constructs an MSF. A partial proof is given below; you complete it.

You will use this counterpart to the safe edge theorem, which we prove for you:

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.

You will now prove the rest by filling in the following:

Claim: When Build-MSF-By-Add-And-Fix terminates, it will produce a Minimum Spanning Forest.

Proof: By induction on the number of edges inspected of this proposition:

> **Proposition P(k):** “After k edges have been inspected (after k iterations), the graph F is a minimum spanning forest (MSF) of the graph G that contains only the edges inspected in line 2”.

Base k=1 (1 pt):

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.

Induction P(k) => P(k+1) (Assume P(k) is true and prove P(k+1) is true. There will be two cases to consider):

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 (2 pts):

If adding the (k+1)-th edge *e=(u, v)* doesn't create a cycle, it means that *e* connects two separate subtrees containing *u* and *v* (one or both which may be trivial trees containing only the vertex). 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 (2 pts):

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 forest is still a MSF.

Conclusion (1 pt): Use the above to conclude that F is correct when the loop terminates and the algorithm returns F:

The loop runs for each edge in |E|. After considering the |E|-th edge, the Proposition says that F 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.

b. (1 pt) Describe an efficient algorithm for finding the maximum weight edge on a cycle, to use in lines 4-5.

Solution: Modify the DFS-Visit-HC code from class on Topic 14B 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 (indicating a cycle)

is found. We call DFS-Visit-HC directly (bypassing the main DFS-Has-Cycle) on one of the vertices at the end of the edge we just added in line 3, since any new cycle must involve this edge.

c. (1 pt) Analyze the time complexity of `Build-MSF-By-Add-And-Fix`, assuming you use the algorithm in (b)**:**

Solution: As we argued in the class on Topic 14B, in an undirected graph it is impossible to see |V| edges without having detected a cycle along the way: hence the cost is O(V) to test lines 4 and 5 simultaneously via the modified DFS-Visit-HC. The outer loop is O(E) so the overall cost is O(*VE*).

Additional comment: It is more expensive than other MST algorithms because the algorithm is not very smart about what edge is added, leading to extra work required to check for cycles and remove maximum weight edges after each edge addition.

---

2. (6 pts) MST on Restricted Range of Integer Weights

Suppose edge weights are restricted as follows. For each restriction,

- Describe a modification to Kruskal's algorithm that takes advantage of this restriction, and
- Analyze its runtime to prove that your modified version runs faster.

Initial Solution for Both Problems: (this reasoning should be included in both a and b, or stated once):
We know that Kruskals algorithm takes O(V) time for initialization, O(E lg E) time to sort the edges, and O(E α(V)) time for the disjoint-set operations, for a total running time of O(V + E lgE + E α(V)) = O(V + E lg E).

If the graph is connected (usually the case when looking for spanning trees!), V = O(E), so we can simplify to O(E lg E).

So, <u>sorting edges is the dominant term. The key is to use the weight restriction to reduce this term.</u>

a. All edge weights are integers in the range from 1 to C for some constant C.

Solution (3 pts):

If the edge weights were integers in the range from 1 to C for some constant C, then we could use counting sort to sort the edges more quickly. Counting sort is O(n+k), where here n=|E| the number of edges to sort and k=C the range of weights. So, sorting would take O(E + C) time, or O(E) since C is a constant.

Now we have a runtime of O(V + E + E α(V)) = O (E α(V)) for connected graphs. Since α(V) is a *very* slow growing function (much slower growing than lg V), this result is faster than the O(E lg V) of the lecture notes.

b. All edge weights are integers in the range from 1 to |V|.

Solution (3 pts):
As above, we can use counting sort to sort the edges in O(E+V) instead of O(E lg E) (replace C with V in the analysis above). V is not constant. However, in a connected graph V = O(E), so this is O(E). Again we get a runtime of O(E α(V)) for connected graphs

---

3. (10 pts) Suppose we change the representation of edges from adjacency lists to <u>matrices</u>. Assume that vertices are represented as integers [1..|V|] and that G.E is a symmetric adjacency matrix where each G.E[*u*,*v*] contains the weight on edge (*u*,*v*) ∈ E, or 0 if there is no edge.

```
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
```

a. What line(s) would have to change in the CLRS version of Kruskal's algorithm (shown), and how? (No need to write the code; just explain the changes required for the matrix representation.)

Solution (2 pts):

Since adjacency lists and matrices represent edges, we need to identify how edges are accessed in Kruskal's algorithm and how the representation change affects this access.

Lines 1-3 only access vertices, so there is no change to this code or their runtime.

Line 4 sorts the edges. This requires having a list or sequence of edge objects to sort. With adjacency lists we could use the list cells as objects to be sorted (presumably the weights are stored in these cells or available in constant time). However, with the matrix there are no explicit objects for edges: we just have weights in a matrix. So we would have to scan the matrix to make a sequence of edge objects to be sorted. Lines 5-9 would not change: line 5 would sequentially process the list resulting from the sort in either case, and there is no change to the near constant time required for lines 6-9, so any change in runtime will be in creating the list of edges to sort.

b. What would be the resulting asymptotic runtime of Kruskal's algorithm on both dense ($E = O(V^2)$) and sparse ($E = O(V)$) graphs, and why?

Solution (3 pts):
Previously, sorting |E| edge objects could be done in O(E lg E) time. With the matrix, we need to account for the time to construct a list of edges. Scanning the matrix to find all non-0 entries requires $O(V^2)$ time. In typical sparse graphs where $E = O(V)$, $O(V^2)$ dominates O(E lg E), so the algorithm would be slower at $O(V^2)$. However, in very dense graphs where $E = O(V^2)$ there would be no asymptotic change (the algorithm was already $O(V^2 \lg V)$.

(In practice, very dense graphs are seldom encountered. If you are eligible, take ICS 422 or 622 to find out more!)

```
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))
```

c. What line(s) would have to change in the CLRS version of Prim's algorithm (shown), and how?

> **Solution (2 pts):** The line "for each v in G.Adj[u]" 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 in the next line we would access weights looking at matrix cell A[u,v], but this is O(1) and won't affect run time.

d. What would be the resulting asymptotic runtime of Prim's algorithm on both dense ($E = O(V^2)$) and sparse ($E = O(V)$) graphs, and why?

> **Solution (3 pts):** The former adjacency list scan now becomes $\Theta(V)$ for the scan of a matrix row. So we have a while loop that is $\Theta(V)$, containing a row scan that is $\Theta(V)$, for $\Theta(V^2)$
>
> There is also an O(lgV) operation Decrease-Key inside the inner loop, but this happens in aggregate *at most* only once for each edge, or O(E lgV).
>
> This gives us $\Theta(V^2)$ + O(E lgV), which we might write as **$O(V^2 + E \lg V)$.**
>
> Since $E=O(V^2)$ we might also write $O(V^2 + V^2 \lg V)$ or $O(V^2 \lg V)$ overall, and we will accept this as an answer. However, we usually include both V and E in complexity analyses of graphs since the rate of growth may depend on edge density (in sparse graphs E = O(V)), so $O(V^2 + E \lg V)$ is better.

4. (6 pts) Divide and Conquer MST

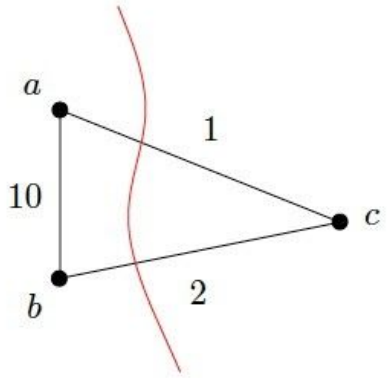
Consider the following divide-and-conquer algorithm for computing minimum spanning trees. The intuition is that we can divide a graph into half, solve the MST problem for each half, and then find a minimum cost edge spanning the two halves. More formally:

Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2) and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Prove that this algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

Solution:

The divide-and-conquer algorithm proposed in the problem will not construct a correct MST. Consider the graph below, with partitions defined as follows $V_1 = \{a,b\}$; $V_2 = \{c\}$.



The recursive calls would be on $G_1 = (V_1, \{(a,b)\})$ and $G_2 = (V2, \{\})$. The divide and conquer algorithm would add edge (a,b) to the MST when constructing the MST recursively on G1. This clearly will not result in a MST for the whole graph, as picking edges {(a,c), (b,c)} will result in a lighter MST of total weight 1 + 2 = 3.

This is *not* a failure of the safe-edge theorem. An edge that is safe for constructing a MST of a subgraph G_1 is not necessarily safe for constructing an

MST of distinct supergraph G. In terms of our example, we cannot claim that (a,b) is a light edge that crosses a cut, since the cut must either be {{a, c}, {b}}, in which case (b,c) is a lighter edge crossing that cut, or {{b, c}, {a}}, in which case (a,c) is a lighter edge crossing that cut.