ICS 311 Spring 2016, Problem Set 09, Topics 15-17

#1. Peer Credit Assignment

1 Point Extra Credit for replying

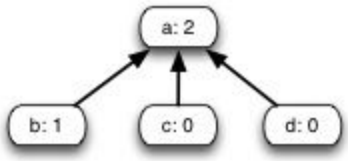
You should have named your group partners and allocated 6 points total across them for class the previous week. If you did not, please email the TA with your points assignment.

---

2. (2 pts) Constructing a Branchy Disjoint-Set Forest

Start with this disjoint-set forest:

Using only the operations Union and Find-Set with rank and path compression heuristics, write the sequence of calls that would create this disjoint set representation from the above nodes:



Solution:

Variations are possible, as shown below. Regardless of variation, to get the ranks shown in the figure, **b** must be the argument to *y* for one call of equal rank, and **a** must be the argument to *y* for two calls, as *y* is the one promoted in line 5 of Link. For example ({x|y} means choose either one, as the set representative will be used):

```
Union(c,a)          // makes a the parent and rank 1
Union(d,b)          // makes b the parent and rank 1
Union({b|d},{a|c})  // makes a the parent and rank 2
Find-Set(d)         // compresses the path
```

Or:

```
Union(d,a)          // makes a the parent and rank 1
Union(c,b)          // makes b the parent and rank 1
```

```
Union({b|c},{a|d})   // makes a the parent and rank 2
Find-Set(c)          // compresses the path
```

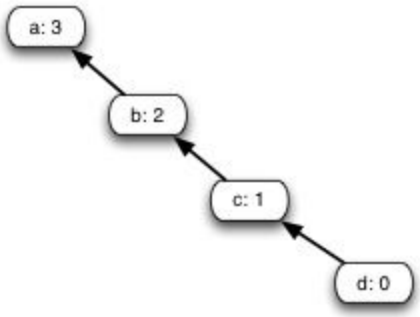
---

3. (4 pts) Linear Disjoint-Set Forest

With binary search trees there was always the possibility that a bad sequence of data would lead to a linear tree. Here we explore whether we need to worry about this with disjoint-set forests.

Start with this disjoint-set forest:

a. (2 pts) Using only the operations Union and Find-Set with rank and path compression heuristics, write the sequence of calls that would create this disjoint set representation from the above nodes:



Solution:

When I created this problem in 2013, I first thought that it is not possible with Union and Find-Set. However, I was assuming that one always calls Union with two different arguments. One can do:

```
Union(a,a)
Union(a,a)
Union(a,a) --> these increase a's rank to 3
Union(b,b)
Union(b,b) --> b's rank is now 2
Union(c,c) --> c's rank is now 1
Union(c,d) --> d will be subordinate
Union(b,c) --> c will be subordinate
Union(a,b) --> b will be subordinate
```

Of course, variations are possible which these operations are done in different orders.

In any case, the point of this problem was to show that you have to set up a very unlikely sequence of operations to get a deep tree!

b. (1 pt) In general, what is required to get a node of rank k?

Solution:
In order for a node to get to rank k, it has to be the second argument *y* to Union with another node of equal rank as first argument *x* exactly *k* times.

c. (1 pt) Why will this never happen when union-find is used as a utility by Connected-Components and Kruskal's algorithm?

Solution:
First I should note that "this" was ambiguous: When originally written the question was meant to refer to the linear tree structure, but once question b was added it seems to refer to nodes of rank k. So we'll need to be flexible in grading based on what students seem to be answering.

"This" = linear tree structure:
Both Connected-Components and Kruskal's algorithm iterate over edges to test whether two vertices are in the same set. We have been assuming simple graphs, in which there are no self loops, so the two vertices will be distinct from each other. Thus we won't get any calls in which the same vertex is given as both arguments. Even if self loops were allowed there would be only one, so such a call would happen only once.

"This" = node of rank k:
It may actually be possible provided that the tree structure is *not* linear, but it would be extremely unlikely. One would need to construct two trees with root of rank *k-1* and union them so that the rank of one is increased to *k*), which in turn would require constructing and unioning four trees with root of rank *k-2* to get the two trees of rank *k-1*, etc., the number of trees growing exponentially with *k*, and the order of calls would have to be just right. Also, due to path compression, at least one path from leaf to root in the tree would have to receive no Find operations on any node in the path to guarantee that the rank actually reflected the height of the tree.

---

4. (6 pts- changed) 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, sorting edges is the dominant term. The key is to use the weight restriction to reduce this term.

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

---

5. (6 pts - changed) Suppose we change the representation of edges from adjacency lists to matrices. *(This is the Kruskal follow-up to what we did for Prim's in class.)*

```
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 Kruskal's algorithm, and how?

Solution (3 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, 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 622 to find out more!)

---

6. (6 pts) Building Low Cost Bridges

Suppose you are in Canada's Thousand Islands National Park, and in one particular lake there are *n* small islands that park officials want to connect with floating bridges so that people can experience going between islands without a canoe. The cost of constructing a bridge is proportional to its length. Assume the distance between every pair of islands is given to you as a two dimensional matrix (an example of such a table for *k*=8 islands is shown below).

Design an algorithm that, given a table such as shown below, determines which bridges they should build to connect the islands at minimal cost. Write down the pseudocode and analyze the runtime of your algorithm. (You are allowed to build on algorithms we have already studied.)

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A	–	240	210	340	280	200	345	120
B	–	–	265	175	215	180	185	155
C	–	–	–	260	115	350	435	195
D	–	–	–	–	160	330	295	230
E	–	–	–	–	–	360	400	170
F	–	–	–	–	–	–	175	205
G	–	–	–	–	–	–	–	305
H	–	–	–	–	–	–	–	–

Solution:

As the above matrix representation should make clear, the problem can be modeled as a graph G=(V,E), with |V|=n vertices (one for each island) and with an edge between every pair of vertices with weight equal to the distance between the pair of corresponding islands. Since we want to build bridges such that every island is reachable, we need to find a connected subgraph T=(V,E') that spans all vertices and for which the sum of weights of all edges E' is minimum. Since all edge weights are positive (the distances between islands), T will be a minimum spanning tree of G (see problem 23.1-7 of CLRS). Thus, to solve the problem, we can apply one of the existing MST algorithms, but must consider the cost of accessing a matrix representation of a graph, or of converting it to an adjacency list representation. We have already addressed the problem of accessing a matrix representation problem for Prim's algorithm in class and Kruskal's algorithm for problem 5 above: see those solutions. In both cases, the algorithms run in $O(V^2)$ time with a matrix. Conversion to an adjacency list representation requires $O(V^2)$ time to scan the cells of the matrix (specifically, 1 + 2 + … + |V| cells) to extract the edges. So, we require $O(V^2)$ time and $O(V^2)$ additional space.

Since students may take different approaches as outlined above we leave it to you to check the pseudocode that accomplishes the above.

---

7. (6 pts) Divide and Conquer MST

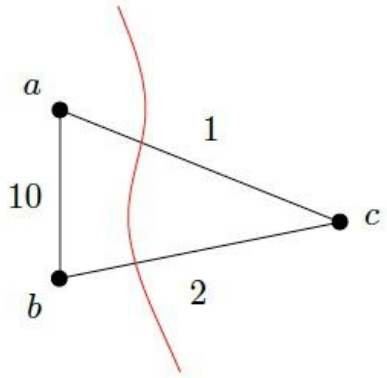
Consider the following divide-and-conquer algorithm for computing minimum spanning trees:

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.