(Sol) HW06 MST and Disjoint Sets

Due: Wednesday, Oct. 18th, 2023 on Canvas

1. (Problem 23-4 in CLRS) Alternative minimum-spanning-tree algorithms (6 points, 3 for a counterexample, 3 for one proof)
   In this problem, we give pseudocode for three different algorithms. Each one takes an undirected connected graph G as input and returns a set of edges T. For each algorithm, you must either prove that T is a minimum spanning tree, or prove that T is not a minimum spanning tree by providing a counterexample. (.TSM ecudorp ton seod smhtirogla eerht eseht fo eno ylno :tniH)
   Answers below. Note that the proof showing T is a spanning tree is optional.

   ```
   1) MAYBE-MST-A(G) {
     sort the edges into non-increasing order of edge weights
     T = E
     for each edge e, taken in non-increasing order by weight do
       if T − {e} is a connected graph
         T = T − {e}
     return T
   }
   ```

   Yes, this does produce a MST.

   (Optional) First, we show that it produces a spanning tree:
   Obviously, the graph that is produced must be connected, since no edge is removed that disconnects the graph. Second, the graph that is produced must be a tree, with no cycles. Proof by contradiction: assume that this algorithm produces a graph that has a cycle. Consider an edge e in this cycle. If we remove this edge, then the graph will still be connected but then the edge cannot be in the graph, since it would have been removed in line 5 of the algorithm.

   Next, we show that it produces a MST.

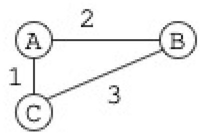
Again, proof by contradiction. Assume T is a spanning tree but not an MST. There must be another spanning tree T' that is an MST. The property of MST of a given graph G is that the cost of MST is less than or equal to the cost of any spanning tree of G. Thus, the sum of weights of edges in T' is strictly greater than that of T. To simplify the notation, let the weight of edge e to be $l(e)$.

If all edges in T have weight no greater than any edge in T', i.e. $\max_{e \in T} l(e) \le \min_{e' \in T'} l(e')$, T would have less cost than T', so not all edges in T can have weight no greater than any edge in T'. In other words, there has to be an edge $e \in T$ that has a higher weight than $\min_{e' \in T'} l(e')$, and $e \notin T$.

MAYBE-MST-A would have removed e if removing e doesn't break the connectivity. Let $e = (u, v)$. Since e is not in T', there should be another path between u and v in T'. If this path contained any edge with higher weight than e, we should be able to replace this edge with e without losing connectivity, so this contradicts that T' is an MST. Thus this path from u to v consists of edges with smaller weights than $l(e)$. However, removing e cannot break the connectivity given that there is the path from u to v with edges with smaller weights than e. This contradicts the algorithm, as the algorithm would have removed e before removing other edges in this path. Thus T is an MST.

```
2) MAYBE-MST-B(G) {
  T = {}
  for each edge e, taken in arbitrary order do
    if T U {e} has no cycle
      T = T U {e}
  return T
}
```

No, this does not produce an MST. Consider the graph:



If we happen to pick the edge (A, B) then (B, C) we will get a spanning tree of cost 5, instead of the optimal 3.

```
3) MAYBE-MST-C(G) {
  T = {}
  for each edge e, taken in arbitrary order do
    T = T U {e}
    if T has a cycle c
      let e' be the maximum weight edge on c
      T = T - {e'}
  return T
}
```

Yes, this does produce a MST.

(Optional) First, we show that it produces a spanning tree:
Obviously, the graph that is produced must be connected, since edge is removed only if it is part of a cycle, so the remaining graph is connected. Second, the graph that is produced must be a tree, with no cycles, since if a newly added edge e created a cycle, then the algorithm removes an edge e' from the cycle.

Next, we show that it produces a MST.
Again, proof by contradiction. Assume T is a spanning tree but not an MST. There must be another spanning tree T' that is an MST. There should be at least one edge $e \in T$ whose weight is greater than any edge e' $\in T$'. (If all edges in T have less than or equal weights than any edge in T' , then T should have less cost than T'.) Let $e = (u, v)$. Since e is not in T', there should be another path between u and v in T', and adding e would have created a cycle. If this path contained any edge with higher weight than e, we should be able to replace this edge with e without losing connectivity, so this contradicts that T' is an MST. Thus this path from u to v consists of edges with smaller weights than $l(e)$. However, this contradicts the algorithm, as the algorithm would have removed e before removing any other edge in the cycle e was in. Thus T is an MST.

(.noitcidartnoc yb foorp yrt :2 tniH)

2. [6 points] (Adapted from CLRS Exercise 21.3-4) Suppose that we wish to add the operation `PRINT-SET(x)`, which takes as input a value x and prints out all of the member in the same disjoint set as x, in any order. Assuming that we are implementing disjoint sets using arrays, design a `PRINT-SET(x)` algorithm that takes time linear in the size of the disjoint set containing x using an additional array of

size n without affecting the asymptotic running times of other operations, where n is the total number of values in the disjoint set.

a. [3 points] Show how this new array is updated on a `UNION` operation, and used in the `PRINT- SET` operation.
   Answer: Our idea is to use the extra array to store a circular linked list of all the nodes in each set. Every time we do a union of two sets, we will combine their linked lists into one larger linked list. To print out all the elements in the set, we need to go through the linked list until we get back to where we started.

b. [3 points] Give pseudocode for your new `UNION` and `PRINT-SET` operations. For simplicity, you do not need to do path compression (though my solution works just fine whether or not path compression is used).
   Answer: This version assumes that the number of vertices in the component is stored as a negative number in root. For example, if vertices 0, 1, 2 are in the same component and 0 is the root, then the parent array will be [-3, 0, 0]. `List[i]` is the index of the node right after node `i`. For example, if two connected components $\{0, 1, 2, 3\}$ and $\{4, 5\}$ are there, and 0 and 4 are the roots, then `List` array could look like this: `[3,0,1,2,5,4]` 3→2→1→0→(3) and 4→5→(4). (Note that this is not the only way `List` array could be. The circular linked list of the first connected component could be 2→3→0→1→(2) and then the `List` array would be `[1,2,3,0,5,4]`.)

   Merging these two circular linked list is a constant-time operation: we just need to swap the root's pointers. When we swap `List[0]` and `List[4]`, then `List` array becomes `[5,0,1,2,3,4]`, 5->4->3->2->1->0->(5). (Using the second example List array, after merging List array become `[5,2,3,0,1,4]`: 0→5→4→1→2→3→(0).)

```
// The only change to this method is setting up
// the initial lists so the time complexity remains O(n)
CreateSets(n)
  for i = 1 to n do
    Parent[i] = −1; // each node is a component by itself
    List[i] = i; // each node is a circular linked list

// This method is unchanged
Find(x)
  if (Parent[x] < 0) // x is the root
    return x;
  else
```

```
    Parent[x] ← Find(Parent[x]); // path compression
    return Parent[x];

// This method merges the disjoint sets and linked lists
union(x,y)
  rootx ← Find(x);
  rooty ← Find(y);
  // merge two components
  // if x's component has more vertices than y's
  if (Parent[rootx] < Parent[rooty]) {
    Parent[rootx] += Parent[rooty];
    Parent[rooty] = rootx;
  else //otherwise
    Parent[rooty] += Parent[rootx];
    Parent[rootx] = rooty;
  // merge the two circular linked lists
  // swap the roots' pointers from two circular linked lists
  tmp = List[rootx]
  List[rootx] = List[rooty]
  List[rooty] = tmp

// To print we only need to go around the list
Print(x)
  tmp = x
  do
    print(x)
    x ← List[x]
  while x != tmp
```

3. [3 points] Accept the assignment on GitHub.
    a. [2 points] Analyze the time complexity of your code.
    b. [1 point] Analyze the space complexity of your code.