CSCI-GA.1170-001/002 Fundamental Algorithms November 27, 2014

Solutions to Problem 1 of Homework 10 (12 (+4) points)

Name: Keeyon Ebrahimi *Due: Wednesday, November 26*

(a) (3 points) Assume directed graph G is acyclic. Show that G has at least one vertex v having no outgoing edges.

Solution: Imagine G as a graph that has all vertexes with outgoing edges, and that after traversing down all Vertexes except for one, we do not have a cycle. We now have to analyze the last non visited Vertex, which we will label as L. Now, if every node has an outgoing edge, then node L will also have an outgoing edge. Because all nodes other than L have already been visited, and L has an outgoing edge, then no matter what, L will have an outgoing node to a node that has already been visited, therefore creating a cyclic Graph. This shows that in order to have an acyclic graph, we need to have a Vertex with no outgoing edges. □

(b) (5 points) Consider the following greedy algorithm for topological sort of a directed graph G: "Find a vertex v with no outgoing edges. If no such v exists, output 'cyclic'. Else put v as the last vertex in the topological sort, remove v from G (by also removing all incoming edges to v), and recurse on the remaining graph G' on $(n-1)$ vertices". If this algorithm is correct, prove it, else give a counter-example.

Solution: Few things to consider

- If we have a vertex with no outgoing edges at any time, we will have an cyclic graph, as shown by $PartA$
- By taking a vertex completely out of an acyclic graph, we will never all of a sudden create a cyclic graph.
- If there is any acyclic section of G, when all of G is acyclic.

With the above statements, we know that the very first removal will not create an cyclic graph out of an acyclic graph, and that it tests for the graph being cyclic or not correctly. By removing this node that has no outgoing edges, we will not be disrupting the tree structure that G has. After removing the vertex with no edges, we now run this algorithm on the newly created tree because if there is any section of G that is acyclic, then the whole graph becomes acyclic. We keep running this test until we remove all Vertexes, which we now know correctly answers if there is any acyclic section or not.

□

(c) (4 (+4) points) It is easy to implement the above algorithm in time $O(mn)$. Show how to implement it in time $O(n^2)$. For **extra credit**, do it in time $O(m+n)$.

Solution: We will be running a modified Topological sort using a DFS.
After the DFS Topological sort, elements with the lowest end time should have no children if we are dealing with a cyclic G. So we run a DFS and sort in reverse finish time order and place them each into a stack. This gives us G in topological sort in running time $O(m+n)$. Now we pop from the stack, giving us the element with the smallest end time, which we will label as L and we see if this child has any children. If it does have children, then we know that we are dealing with an cyclic G, for in an acyclic G, the vertex with the smallest end time will have no children. If L does not have any children, we remove this L Vertex from G, and also remove all incoming edges. This process will happen in constant time because we are storing G in a $2x2$ array.

We have now accurately implemented the initial step of the algorithm in time $O(m+n)$. After a successful removal of L, we once again pop from our stack, giving us the new Vertex with the least ending time. With this vertex we run our constant time checks and repeat.

To make this happen, we need to run our topological sort and place these all into a stack, which will take $O(m+2n)$. This is because the topological sort takes $O(m+n)$ and placing in the stack takes time $O(n)$. We then pop from the stack, make checks on children and update the problem base on this info, which only takes constant time.
Our running time is now $O(m+2n)$, and we are also going to pop from the stack $0(n)$ times, meaning our total running time for this algorithm will be $O(m+3n)$, which is equivalent to $O(m+n)$ □

CSCI-GA.1170-001/002 Fundamental Algorithms November 27, 2014

Solutions to Problem 2 of Homework 10 (6 points)

Name: Keeyon Ebrahimi *Due: Wednesday, November 26*

Recall, MST finds a spanning sub-tree T of the original graph minimizing the sum of edge weights in T: $\sum_{e \in T} w(e)$. Consider a related problem MST$'$ which attempts to find a spanning sub-tree T' of the original graph minimizing the maximum edge weight in T': $\max_{e \in T'} w(e)$. Show that the solution T to MST is also an optimal solution T' to MST$'$. Is the converse true as well?

Solution:
In this problem it is very important to look at both the cut property and the cycle property.

1. **Cut Property:** This states that whenever you put a line through the graph, also called a cut, of all edges that this cut touches, if there is an edge that is strictly smaller than all the other edges that the cut touches, then that edge must belong in the MST.

2. **Cycle Property:** This states that for any cycle within G, if there is an edge that is strictly larger than all the other edges, then this edge should not be in the MST.

We can see that for the Original MST, we are looking for the minimal sum, meaning we follow the cut property picking the edge with the least weight until we have our completed MST.
We can also see that for the related MST', we will continuously be following the Cycle Property, eliminating edges with the greatest weight.

If we can show that these two properties are actually doing the same thing, we can show that each process will be a solution to the other.

Lets imagine we have G, that is split into two sub Graphs, A and B, which have no overlap with each other. Graph A and graph B are completely disjoint. Lets also say that we have created this division of A and B to have only two edges connecting graph A and B as well.

So we have graph A that is a subset of G. We have B that is a subset of G. A and B are completely disjoint, and also have 2 edges connecting A to B.

These two edges are not going to have the same value as well, so one of the edges will be larger, l, and one will be smaller, s.

Now if we use the cut property through s and l, we will be selecting the smaller value, so we will be selecting edge s for our MST.

Now lets use the cycle property instead of the cut property. With the cycle property, we will

examine s and l and eliminate l. The cycle property will select s after eliminating l.

As we can see, for this snapshot, both the cut property and the cycle property will both be selecting the same edge for our finished MST and our MST'. If we repeat this process throughout the entire G, we will also eventually be selecting all of the same edges for MST and MST', giving the use of each the same optimal solution

□

CSCI-GA.1170-001/002 Fundamental Algorithms November 27, 2014

Solutions to Problem 3 of Homework 10 (10 points)

Name: Keeyon Ebrahimi *Due: Wednesday, November 26*

(a) (4 points) Assume that all edge weights of an undirected graph G are equal to the same number w. Design the fastest algorithm you can to compute the MST of G. Argue the correctness of the algorithm and state its run-time. Is it faster than the standard $O(m + n \log n)$ run-time of Prim?

Solution:
I claim that with all edges weighing w, then any tree that includes all Vertexes will all have the same total weight, thus will be a MST. If this claim is true, a correct algorithm would be to traverse through G and add all non visited vertexes and the edge that connected them into our MST. To do this, we would just run a Depth First Search and traverse through G, adding all non visited Vertexes and the Edges connecting to them into our MST. With this algorithm, we will only be traveling to each vertexes edges. If we have our graph stored in an adjacency list, this search will have a running time of $O(m+n)$. **This $O(m+n)$ algorithm will have a faster running time than Prim's Algorithm.**

Now we must prove that this algorithm will give the correct result. With all MST's, we know that we need to be able to connect all vertexes, only visiting each vertex once, which means that we will always have a total of $V - 1$ edges. If all edges weigh the same, and all spanning trees have $V - 1$ edges, then we know that all spanning trees in this example will have the total weight of $w * (V - 1)$. This means that all spanning trees of G will be a MST. As long as we create a spanning tree, we will also have a MST. A DFS will always give us a tree spanning all vertexes, and because all spanning trees in this example are a MST, we know that this algorithm will give us a correct solution. □

(b) (6 points) Now assume the all the edge weights are equal to w, except for a single edge $e' = (u', v')$ whose weight is w' (note, w' might be either larger or smaller than w). Show how to modify your solution in part (a) to compute the MST of G. What is the running time of your algorithm and how does it compare to the run-time you obtained in part (a) (or standard Prim)?

Solution:
My above solution will not work as is, because if w' is less than w, we have the chance of skipping over it in a regular old DPS. Also, if w' is more than w, now we have the chance of including this in our solution, yet both of these situations will not give us a correct MST.

Instead of running a DFS, we could now instead run a BFS, and instead of just selecting the first edge we see, we know have to compare all the edges and select the edge with the minimum edge weight. With this, we would now include w' whenever $w' < w$, and we would

also skip over w' if $w' > w$, giving us the correct MST

This new algorithm would now have a different running time. Now when we hit a new vertex, instead of just going edge to edge until we hit an edge that visits a new vertex, we now have to make sure we go through every edge and select the one with the minimum weight.

As we can see though, this is exactly what Prim's algorithm does. It goes to a Vertex, find the minimum weight edge, and includes it if a cyle is not created. We are doing the exact same thing here now, which we have to do in order to make sure we handle w' correctly.

Adding one w' element that is different than w now forces us to check the weight of all edges, making us now have to use Prim's algorithm which has the $O(m + n\ log\ n)$ edges. $\square$

CSCI-GA.1170-001/002 Fundamental Algorithms November 27, 2014

Solutions to Problem 4 of Homework 10 (16 points)

Name: Keeyon Ebrahimi *Due: Wednesday, November 26*

Assume all edge weights in G are integers from 1 to w.

(a) (8 points) Show how to modify Prim's algorithm to achieve running time $O(m+nw)$. Hence, if $w = O(1)$, you get optimal time $O(m+n)$.

Solution: A modification to give the $O(m+nw)$ running time is based on the fact that we now know the minimum possible value of our edges. The modification would that whenever we visit an edge with weight 1, which is our minimum possible value, and this edge keeps our solution acyclic, we select that edge because we know that this is the minimum an edge can possibly weigh, giving us the correct edge without having to span through the rest of the edges.

In this example, if $w = O(1)$, then all of our weights will be the same at 1, and this will make us always use the first edge we will see, which will give us the running time $O(m+n)$, as explained in Problem 3*a*. □

(b) (4 points) Now assume $w = n$, so that the previous solution in part (a) is no longer faster than standard. Show how to modify Kruscal's algorithm instead of Prim's, so that it now takes time $O(m + n \log n)$, instead of $O(m \log n)$.

Solution:
In regular Kruscal's algorithm we have three main steps

1. **Make n sets. (One for each vertex):** $O(n)$
   Self Explanatory

2. **Order edges by weight:** $O(m\ log\ n)$
   Sorting usually has a running time of $O(m\ log\ m)$. In a regular Kruscal's algorithm, this will reduces down to $O(m\ log\ n)$ and this is why. With G, the maximum edges we can have is when each vertex have an edge to each other vertex. This would give us v^2 edges. This means that we can reduce $O(m\ log\ m)$ down to $O(m\ log\ n^2)$. This equals $O(m\ 2 * log\ n)$, which equals $O(m\ log\ n)$

3. **Iterate and fill MST:** $O(n\ log\ n)$
   Fill out Iterate through edges in the previously sorted order. For each edge, check if the connecting vertexes are in each others set. If not add the newly discovered vertex to the MST. We will iterate n times, (connecting n vertexes to the final spanning tree), and each check runs in $O(log\ n)$ time because the depth of the Set Tree is at maximum $log_2(n+1)$. Then if it is the case where it is disjoint, we add this value into our MST, which runs in constant time. This means that total, we have a running time of $O(n\ log\ n)$

So, when we add the running time of these three steps for regular Kruscal's Algorithm, we get

$$O(n + m\ log\ n + n\ log\ n)$$

This then can be reduced to:

$$O(m\ log\ n)$$

The dominating factor is the $O(m\ log\ n)$, which comes from the sorting from step 2.

We must now look at our problem where $w = n$. With this information, we can improve our search step. With this limited range on our values, we can now get away from any comparative sort taking $O(m\ log\ n)$ time, and we can now run the Count Sort, which will give us a running time of $O(m + n)$.
After we do this, we notice that our complete running time of steps 1, 2, and 3 becomes

$$O(n + m + n + n\ log\ n)$$

This can then be reduced to:

$$O(m + n\ log\ n)$$

□

(c) (4 points) What is the largest w for which you can still maintain the $O(m + n \log n)$ run-time in part b? In particular, can you tolerate $w = n^2$? $w = n^3$?

Solution:
In order to keep ourselves under the $O(m\ log\ n)$, running time, we need to keep the sorting time under the running time of $O(n\ log\ n)$. If we can keep **Step 3** from *part b* as the dominating part of the algorithm, then we can keep the running time of $O(m + n\ log\ n)$. If the sorting from **Step 2** ever goes over $O(n\ log\ n)$, then we can no longer run in $O(m+n\ log\ n)$

As long as our range is slightly above n, then we can still get $O(m + n)$ with the Counting sort. With $w = n^2$ or $w = n^3$, the counting sort will now give running times of $O(m+n^2)$ and $O(m+n^3)$, which boils down to $O(n^2)$ and $O(n^3)$ respectively. At this point, our Counting Sort becomes obsolete, and it will be best to use a comparative sort running in $O(m\ log\ n)$ time. When this is the case, we now go back to the total running time of $O(m\ log\ n)$, **making both $w = n^2$ and $w = n^3$ not able to maintain the running time of** $O(m + n \log n)$ □