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

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

*Name: Sahil Goel* *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:** Suppose that every vertex have outgoing edges. In this case, whenever we will reach any vertex, we will have other nodes where we can visit from this node. Therefore we will always be able to visit some node from any node. Hence we won't reach the leaf node ever since there will be no leaf node. This can only be true if the graph has infinite nodes or graph has a cycle or at least one of the vertex have no outgoing edges. Since we know that graph doesn't have infinite nodes and the graph is acyclic one of the vertex must have no outgoing edges. Only then graph can be acyclic otherwise we will keep traversing the same nodes which implies that there will be cycle in the graph.

□

(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:** Yes the greedy algorithm is correct. Proving it using local swap method
Suppose there is an optimal solution or topological sorted nodes $Z_0 = \{\ v_1, v_2, \ldots, v_n\}$

Local Swap

1. According to the greedy approach we will replace $v_1$ with the node say $v_1'$ which has no outgoing edges. Since $v_1'$ doesn't have any dependency hence it can be put at the first position in the topological sort. Also since $v_1$ occurs first in the topological sort in optimal solution it also shouldn't have any dependency. Therefore we can easily replace the $v_1'$ with $v_1$ and then also the solution will be topological sort. Since there will be o backward edge as there is no dependency of $v_1$ and $v_1'$

2. Now since we have removed all the incoming edges of $v_1'$, all the nodes which have dependency on $v_1'$ can now be completed. Say we choose $v_2'$ from greedy method and swapping it with $v_2$. We can easily swap it since $v_2'$ occurs in graph only if it has no dependency now. Hence we can easily choose $v_2'$ in this step and continue forward.

Continuing in this manner till all the nodes are processed will give the topological sort of all the nodes in the graph.

$\square$

(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:** O($n^2$) Algorithm
Initializing the graph using adjacency lists

```
TopologicalSort(G)
{
1. Initialize in degrees
1(a). For each v in graph G
Initialize in degrees to 0
1(b). For each v, traverse its adjacency list and increment in-degrees of all the nodes in adjacency list by 1.

2. While there are vertices remaining(Vertices with positive in-degrees)
2(a). Now choose the node which has zero in-degree. If no such vertex, output "Cyclic" and exit. else print this vertex
2(b). Assign in-degree -1 to the current node
2(c). Traverse the adjacency list and decrease in degrees of nodes by 1.
2(d). Repeat this step

}
```

Total complexity (n & m are nodes and edges respectively):
(a). Initializing in-degrees: $|m|$
(b). Finding vertex with zero in-degree. Since there are n vertices and each will take O(n) time to search, total time will be O(n*n)= $O(n^2)$
(c). Reducing in-degrees of all nodes by 1 = $O(m)$
(d). Output and mark vertex = $O(n)$
Total complexity will be $O(n^2 + m + m + n)$. Since m is always $\leq n^2$, therefore
Total Time complexity = $O(n^2)$

**Extra Credit**

Fundamental idea: We will use queue to store the vertices with in-degree zero so that we don't have to search for vertices with in-degree 0.

```
TopologicalSort(G)
{
1. Initialize in degrees
1(a). For each v in graph G
Initialize in degrees to 0
1(b). For each v, traverse its adjacency list and increment in-degrees of all the nodes in adjacency list by 1.

2. Initialize Q={}
3. Search for a vertex with in-degree zero. If no such vertex output "cyclic" and exit.
Else Enqueue this vertex.

4. While vertices remaining and Q is not empty
4(a). Dequeue Q and print the vertex
4(b). Assign in-degree -1 to the current node
4(c). Traverse the adjacency list of this node and decrease in degrees of nodes by 1. If any node in-degree becomes zero enqueue it
4(d). Repeat this step

5. If all the nodes have been traversed(i.e. indegree = -1) then processing is complete else output "cyclic"
}
```

Total complexity (n & m are nodes and edges respectively):

(a). Initializing in-degrees: $|m|$ = O(m)

(b). Finding vertex with zero in-degree. Since dequeuing only takes O(1) time and there are only n vertices, total time will be O(n)

(c). Reducing in-degrees of all nodes by 1 = $O(m)$

(d). Output and mark vertex = $O(n)$

Total complexity will be $O(n + m + m + n)$. Total Time complexity = $O(m + n)$

$\square$

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

Solutions to Problem 2 of Homework 10 (6 points)

*Name: Sahil Goel* *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:** T: minimum spanning subtree
T': spanning tree with minimizing the maximum edge
Using Cycle Property
Suppose that optimal solution for T is not optimal for T'. Therefore there exists an edge E in T whose weight is larger than the maximum edge in T' which is E'. Suppose E connects (u,v). Now in T', we connect this edge E, which creates a cycle in $T'$ between u and v. Now since we know that this edge was in T, this can't be the maximum edge of the cycle because otherwise we would have chosen a different edge from (u,_) or (_,v) or (u,v) which had the smallest weight. Therefore it means that there is another edge in the cycle which is maximum whose value is larger than the Edge (u,v). Therefore this is an contradiction and E (u,v) is not the maximum edge and maximum edge would not occur from the solution of T because we always choose the minimum possible edge among all the options. If we continue in the same manner we can prove that at every step while building the optimal solution for T, it will be an optimal solution for T' too.

The reverse is not true because we can swap any number of smaller edges in the graph which are smaller than the maximum, still the solution will be optimal for T' but not for T. Example, suppose there are 4 nodes and below are their edge weights A-B : 24
A-D : 25
A-E : 10
B-C : 9
B-D : 11
B-E : 30
C-D : 8
D-E : 28
Now optimal solution for T is (E-A, A-B, B-C, C-D)
It is also an optimal solution for T' with maximum edge AB of weight 24
Now if we swap edge B-C with B-D, it will no longer be optimal solution for T, but it will still be optimal for T', since the maximum edge doesn't change but sum of the edges now increases. Therefore vice versa is not true.

□

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

# Solutions to Problem 3 of Homework 10 (10 points)

*Name: Sahil Goel* *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:** Since all the edges have same weights, we can do DFS traversal on it and output the path from starting node to ending node. In DFS, every node will be encountered only once and since there are all equal weight edges, we can add any edge between any two nodes. Also since total number of edges will be n-1 connecting n vertices and each edge has weight w, total weight of the MST will be w*(n-1). Also since there can be a cycle in this graph, we will just ignore that and handle that case in code(just consider new white nodes and leave gray nodes untouched). Total time complexity will be O(n+m)

```
MST(G)
{
DFS(G)
}

DFS(G)
For all u ∈ v color[u] = white
parent[u] = nil
end for
for all u belonging to v (u ∈ v)
if(color[u] = white)
DFS visit(G,u)
}

DFS-Visit(G,u)
{
color[u] = gray
for all v∈ adj[u]
{
if color[v] = white
parent[v] = u
Add Edge(u,v) to MST print("Edge u,v")
DFS visit(G,v)
}
Color[u]=black
end for
```

```
}
```

Proof
Since DFS always processes only white nodes, it will never process an edge which connects already in progress or processed node. Therefore it will always output such path in which at least one new node will be present and print that edge. Since there are only n nodes, total number of edges that will be added are n-1. Also since the weight of each edge is same, the total weight will be w*(n-1) and hence it will be MST for such graph G

□

(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:** This can also be done in O(m+n) time.
Initially we will find the edge with different weight w'. If w' < w, we will add this edge to the spanning tree (suppose it is between u and v), and make color of u & v gray an call DFSvisit(u). Otherwise we will just call DFS and follow the same procedure. If w(E) (i.e. weight of any edge E)=w' at any point we won't print that edge. If at the end any node is still white we will add this edge, since there is no other path to this node except this

```
MST(G)
{
For all u ∈ v
color[u] = white
parent[u] = nil
end for
Find w, w', u and v
if w'<w
Add edge (u,v) to MST
Color[v]=gray
DFSvisit(G,u)
else DFSvisit(G,u)
if(all nodes not traversed, add edge(u,v) to the MST)
}

DFS-Visit(G,u)
{
color[u] = gray
for all v∈ adj[u]
{
```

```
if color[v] = white
parent[v] = u
if ($w[E] \neq w'$)
add edge to MST
print(”Edge u,v”)
DFS visit(G,v)
}
Color[u]=black
end for
}
```

It is faster than Prim’s algorithm and equal to algorithm in part A.

□

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

# Solutions to Problem 4 of Homework 10 (16 points)

*Name: Sahil Goel* *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:** The running time of prim's algorithm is determined by the speed of queue operations(Extract-min and Decrease-key). Since the keys in the priority queue are the edge weights and there are restrictions on the edge weights we can implement it more efficiently and reduce the running time. We can take an array Q [0..w+1] of w+1 slots where each slot will hold a doubly linked list of vertices with that weight as their key. (w+1) holds the keys with weight infinity. Then Extract min will now only take time O(w) since we will just output the first non empty slot and Decrease key will also take O(1) time, since we will remove it from the list and insert it into the front of the list indexed by new key. Since there will be n-1 Extract min operations hence total complexity for this step will be O(n*w). Decrease key operation will be performed on m edges therefore total complexity of that step will be O(m). Hence total complexity of the solution will be O(m + nw)

```
Algorithm
Prim's
{
Create array arr[w+1]. Add all nodes to the doubly linked list arr[w+1] with keys=Infinity
key[s] = 0 for some arbitrary s
while all vertices not processed
do
u = Extract-Min(Q) by scanning array from left to right and extracting first non empty element
For each v ∈ Adj[u]
do if v not processed & w(u,v) < key[v]
then key[v] = w(u,v) % Decrease-Key by removing it from current list and adding to new list
$\pi[v] = u$
}
```

Total time complexity = n*T(Extract-min) + m*T(Decrease-key)
T = O(nw+m)

If weight can only have 1 value, then this algorithm will take only time O(n+m) since we can output any of the edge between any two nodes

□

(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 Kruskal'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 kruskal's algorithm we can sort the edges using radix and counting sort. It takes time O(dm) here d=c(proved it below). Hence time taken to sort those edges will be O(m). Also, for adding new edges to the MST, instead of using simple disjoint data structure, we can use modified version of it. We will still use different linked lists for each set, but every node in linked list will now have 2 pointers. 1 pointer will points to the head of the node and 2nd pointer points to the next element in the linked list. Now, find operation will take O(1) time. For Union, we will follow the same approach, by combining smaller set to larger set. We can store this information in the head only i.e. total number of elements in this set. After combining two sets, we still need to change the head of all the smaller nodes list. Since maximum number of elements in any node will be $\log n$, total time this step will take is O($\log n$).
T(find) = O(1)
T(Sort) = O(m)
T(Union) = O($\log n$)

Total time : O(T(Sort) + m*T(find) + n*T(union))
Total time : O(m + m + $n \log n$)
T = O(m + $n \log n$)

Changes from original Kruskal Algorithm
1. Sort edges using counting and radix sort in time O(dm), where d is s.t weights of edges vary between 0 to $m^d$-1. Since range is from 1 to n
$m^d$ = w
d= $\log n / \log m$
Since m will be between n-1 to $n^2$, d will be a constant c. Therefore Total time to sort is O(c*m)= O(m)
2. Using doubly linked list for find and union data structure. Every node stores head as the top of that set and tail as the next node in the list. Time taken to find is O(1) and time taken to union 2 lists of size n1,n2 is O(min(n1,n2)). Since n1,n2 $\leq \log n$, total time is O($\log n$)/ . Since Find will be called maximum m times and union will be called n-1 times total time of this step will be O(m + $n \log n$)

$\square$

(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:** Since from previous part we know that, only sorting step will have a different time depending on the range of the input. Recall
$m^d$=$w$

d = $\log w/\log m$
Since maximum m=$n^2$
d = $\log w/(2 * \log n)$
d will be constant for any w = $n^c$
hence range can be from 1 to $n^c$ for any c. But range should not be a increasing function of n. If that's the case then the total complexity will increase
For 1 to $n^2$ : d=1, time for sorting = O(m)
For 1 to $n^3$ : d=3/2, time for sorting = O(3m/2)
for 1 to $n^c$ : d = c/2, time for sorting = O(cm/2)
Until c is not an increasing function of n, we can tolerate it without affecting the overall complexity

$\square$