Homework #7
Algorithms I
600.463
Spring 2017

Due on: Tuesday, April 18th, 11:59pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: On Gradescope, under HW7
Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

April 18, 2017

Problem 1 (20 points)

Let $G = (V, E)$ be a directed graph. Vertex $a \in V$ is a *central* vertex if for all $b \in V$ there exists a path from a to b. Design an algorithm to test whether graph G has a *central* vertex in $O(V + E)$ time. Prove the correctness of your algorithm and analyze the running time.

Answer: Problem 1

We are looking for a vertex that can reach all the other vertices through a directed graph. It is possible that there are no central vertices, one central vertex, or multiple central vertices in a graph. In class, we learned a very easy method but it isn't the best when we are dealing with graphs that are overly extensive or very large. We could perform a depth-first search or breadth first search on each vertex and then keep track of which vertex can reach all the other vertices. This will find the central vertex eventually but not in the complexity we want (this approach is $O(n^2)$. We can fix this to be done in $O(V + E)$ time. In chapter/section 22.5 of CLRS, the concept of strongly connected components can help in this regard. By decomposing a directed graph into its strongly connected components, we create a directed and acyclic graph which means that we can find the central vertex easily, by checking if it contains a single source vertex. According to the book, once a vertex (and all its descendants of that vertex) is reached, the central vertex is the one that simply takes the longest time to traverse, or the maximum finish time (in the depth first search traversal). To do this, two recursive calls can be made implemented with a depth first search algorithm.

To prove why the above algorithm works, we can divide our problem into two cases, the first of which is when a central vertex does not exist. In which case the 'verification' part of our algorithm will fail and given us the correct answer.

In the other case, where a central vertex does exist, we can argue that whenever we make a DFS call of a node u (marked as (1) in our algorithm) it implies that none of the vertices that have been encountered ('visited') so far could be central vertex, since none of them managed to reach the vertex u. Therefore, if a central vertex exists it has not been encountered so far. Further, mantaining our assumption of the existence of said central vertex, one of the remaining vertices must be a central one. Since this is the last call, the vertex making this call can reach every remaining vertex (including, trivially, itself) and is therefore a central vertex itself.

Our DFS functions executes once for every node, because of the constraint placed by visited sets. Each function call is also limited in loop iterations by the number of edges of that specific node. Making the complexity of each DFS function O(V + E) spread over all possible calls. The first outer loop runs O(V) times. We assume our set opertaions to be O(1) easily represented by an array if the nodes are represented by first —V— natural numbers, or a hashmap otherwise. Making their inidividual operations O(1) and the final comparison O(V).

This gives us an overall time complexity of O(V + E).

We perform a simple DFS on the graph. We argue that if at least one vertex exists that can visit every other node, the last vertex to initiate a DFS call must be one of them.

We assume a set of nodes V and an adjacency list containing the edge data of the graph.The adjacency list representation can be created easily as follows from the given set E:

```
1: function ADJACENCY-LIST(v, G)
2:     adj = list of lists
3:     for all nodes u in V do
4:         adj[u] = new empty list
5:     end for
6:     for all edges u, v in E do
7:         adj[u].append(v)
8:     end for
9: end function
```

Here is another pseudocode using the strongly connected component graph algorithms from CLRS (Ch. 22).

```
1: function STRONGLY-CONNECTED-COMPONENT(G, v)
2:     DFS(G)
3:     do DFS traversal, find last visited vertex.
4:         set that equal to v
5:     G^T ← TRANSPOSE(G)
6:     DFS(G^T)
7:     G ← new GRAPH( )
8:     return G, v
9: end function
```

```
1: function CHECK-FULL-TRAVERSAL(G, v)
2:     WHILE(there exists edges in T = (s, v))
3:     if every vertex is reachable from v.
4:         return G, v
5: end function
```

My Implementation/Algorithm:

```
visited = set() // set of visited nodes, initially empty
DFS(u):
    if u is in visited:
        return u
    insert u into visited
    for every node v in the adjacency list of u:
        DFS(v)
last = NULL
for every node u in V:
    if u is not in visited:
        last = u
        DFS(u)

// now we verify if it is indeed a 'central' vertex
visited-verify = set()
DFS-verify(u)
    if u is in visited-verify:
        return
    insert u into visited-vertify
    for every node v in the adjacency list of u:
        DFS-verify(v)
    DFS-verify(last)
if (V == visited-verify):
    G has at least one central vertex, last
else G does not have a central vertex
```

Problem 2 (20 points)

A "friendly" Airline has n flights [1]. In order to avoid "re-accommodation", a passenger must satisfy several requirements. Each requirement is of the form "you must take at least k_i flights from set F_i". The problem is to determine whether or not a given passenger will experience "re-accommodation". The hard part is that any given flight cannot be used towards satisfying multiple requirements. For example, if one requirement states that you must take at least two flights from $\{A, B, C\}$, and a second requirement states that you must take at least two flights from $\{C, D, E\}$, then a passenger who had taken just $\{B, C, D\}$ would not yet be able to avoid "re-accommodation".

Your job is to give a polynomial-time algorithm for the following problem. Given a list of requirements $r_1, r_2, \ldots, r_m$ (where each requirement r_i is of the form: "you must take at least k_i flights from set F_i"), and given a list L of flights taken by some passenger, determine if that passenger will experience "re-accommodation".

Specifically, you just need to show how this can be reduced to a network flow problem and assume there is a given polynomial-time blackbox algorithm solving the flow problem. Prove that your reduction is correct.

[1] Any relation to actual airlines of similar name is purely coincidental.

Answer: Problem 2

We can use a flow network designed as follows. The network is composed of two layers of nodes, apart from the source and sink nodes. Layer one contains a node for every flight. Layer two contains a node for every requirement (r).

Proof: If an assignment exists, we can use it to achieve the required flow. We of course induce the maximum possible flow from source to the sink. In a given 'valid' assignment each flight will be a part of at most one of the given sets, and also be a part of the list L. We induce a flow of 1 from source to the flight's node in layer 1. We also induce a flow of 1 (carrying this same incoming 'fluid') from the flight's node in layer 1 to the assigned requirement's node in layer 2. Since it is a valid assignment each node in layer 2 will receive the required flow of k for that requirement. Making the max flow achieved equal to the sum over all k's.

The complexity is the same as the max flow algorithm we employ, the best of which we know to run in polynomial time.

BLACKBOX ALGORITHM:
for every flight f in L
    add an edge from $source$ to the corresponding node of the flight in the first layer with capacity 1
for every requirement r:
    for every flight f in the set F of the given requirement:
        add an edge from the flight's node in the first layer to this requirement's node in the second layer with capacity 1
    add an edge from the requirement's node in the second layer to the sink node with capacity k (the minimum number of flights we must take)

Run max flow from source to sink
if maxFlow == sum of k's accross all requirements:
    no re-accomodation
else re-accomodation