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 17, 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:

First, we notice that the directed graph G can be cyclic. Is there any method we can get rid of all the cycles? Let's consider one possible algorithm described as following.

1. Use Kosaraju's algorithm to find all strongly connected components of the given graph G.
2. Build a new graph G^* by shrinking each SCC of G into one representative vertex and keeping the edges between SCCs.
3. If there is more than one vertex that has no incoming edges, return FALSE; else, return TRUE.

Running Time:
Assuming that the graph is presented by adjacency list, Kosaraju's algorithm runs in $O(|V| + |E|)$ time. Building the new graph G^* and finding outgoing-only vertices take $O(|V| + |E|)$. In total, the time complexity is $O(|V| + |E|)$.
Correctness:
I eliminated some details here. Briefly, let's consider the graph G^*. Clearly, G^* is a DAG. DAG has at least one vertex with no incoming edges (we can use proof by contradiction, eliminated here). So the only case left, one vertex with no incoming edges is actually the central vertex. This is because all other vertices have at least one incoming edge and can be reached by one of vertices. If there are more than one such vertex, e.g. two vertices, one cannot be reached by the other, which contracts the definition of 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.
Answer:
First, we can create a bipartite graph where on the left we have one node for each requirement r_i, and on the right we have one node for each flight taken by the passenger. Put an edge between requirement r_i and flight j if $j \in F_i$. give all these edges a capacity of 1. Now let's create a source with an edge of capacity k_i to each

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

node r_i on the left, and create a sink with an edge of capacity 1 to it from each flight node.

Claim 0.1. *The passenger can avoid "re-accommodation" if and only if there is a flow of value $k = \sum_i k_i$.*

Proof. First, if the passenger can avoid being "re-accommodated" , then there must be a flow of value k, since we flow k_i units into each node r_i, split this flow into k_i pieces of size 1 going to the k_i flights used to satisfy the requirement, and finally send all that flow to the sink. At most 1 unit of flow will be on any edge into the sink because we are guaranteed that no flight is being used to satisfy more than one requirement. In the other direction, if there is a max flow of value k, then there must be one in which all flows are integral (by the Integral Flow Theorem). Such a flow must therefore, for each requirement r_i, have k_i edges exiting it with flow equal to 1. In addition, no two flow-carrying edges from different requirements go to the same flight node. Therefore, these edges with flow 1 indicate a legal way to satisfy all the requirements. □