Homework #6
Introduction to Algorithms
601.433/633
Spring 2020

Mou Zhang
Due on: Tuesday, April 21st, 12pm

1 Problem 1 (15 points)

Let $G = (V, E)$ be an undirected graph with *distinct non-negative* edge weights. Consider a problem similar to the single-source shortest paths problem, but where we define path cost differently. We will define the cost of a simple s-t path $P_{s,t} = \{e_1, e_2, \ldots, e_k\}$ to be

$$c(P_{s,t}) = \max_{e \in P} w_e$$

The cost of a path is now just the largest weight on that path, rather than the sum of the weights on the path. Give an algorithm that takes as input an undirected graph $G = (V, E)$ with *non-negative edge weights*, and a vertex $s \in V$, and computes the path from s to every other node in G with the least cost under cost function $c(\cdot)$. That is, for each $t \in V$, find a simple path $P_{s,t} = \{(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, t)\}$ connecting s to t, such that, letting S denote the set of all simple paths connecting s and t, $c(P_{s,t}) = \min_{P \in S} c(P)$.

Your algorithm should run in $O(|E| + |V| \log |V|)$ time. Prove the correctness of your algorithm and its runtime. Hint: note the similarities between Dijkstra's algorithm and Prim's algorithm.

Solution:

My algorithm is nearly same as the Dijkstra algorithm. The only difference is how I define the length of path in the shortest path problem.

Algorithm 1 Dijkstra_with_min_max_weight (G, w, s)

```
1: procedure DIJKSTRA(G, s)
2:     S is a set, S ← ∅
3:     Q is a Fibonacci heap, Q ← G.V
4:     while Q ≠ ∅ do
5:         u ← EXTRACT − MIN(Q)
6:         S ← S ∪ {u}
7:         for each vertex v ∈ G.Adj[u] do
8:             RELAX(u, v, w)
9:         end for
10:    end while
11: end procedure
12: procedure RELAX(u, v, w)
13:    returnMIN(v.d, MAX(u.d, w))
14: end procedure
```

Proof to Correctness

Proof. Let's assume the largest value from s to vertex u is $\phi(s, u)$.
The invarient we need to prove is that at the start of each iteration of the while loop of lines 4–10, $v.d = \phi(s, v)$ for each vertex v ∈ S.
If the condition above is true, then for every vertex u ∈ S, We have $u.d = \phi(s, u)$ at the time when u is added to set S. Once we show that $u.d = \phi(s, u)$, we rely on the upper-bound property to show that the equality holds at all times thereafter.

Initialization: Initially, $S = \emptyset$, and so the invariant is trivially true.

Maintenance: We wish to show that in each iteration,$u.d = \phi(s, u)$ for the vertex added to set S. We gonna prove our method by contradiction. Let u be the first vertex for which $u.d \neq \phi(s, u)$ when it is added to set S. We shall concentrate on line 4, the beginning of the while loop for the purpose of contradiction that $u.d = \phi(d)$ at that time by examining a shortest path from s to u. It is obvious that $u \neq s$ because s is the first vertex added to set S and $s.d = \phi(s, s) = 0$ at that time. Because $u \neq s$, we also have $S \neq \emptyset$ just before u is added to S. Another thing is that there must be some path from s to u for otherwise $u.d = \phi(s, u) = \infty$ by the no-path property, which violated out assumption that $u.d \neq \phi(s, u)$. Since there is at least one path from s to u, there exists a shortest path p from s to u. Before adding u to S, path p connects a vertex in S, namely s, to a vertex in $V - S$, namely u. Let us the first vertex along p such that $y \in V - S$, and let $x \in S$ be

$y's$ predecessor along p. Thus we can decompose path p into $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, in which both p1 and p2 can have have no edges.
Now we have a claim that $y.d = \phi(s, y)$ when u is added to S. Since $x \in S$ and u is the first vertex for which $u.d \neq \phi(s, u)$ when it is added to S, we have $x.d = \phi(s, x)$ when x is added to S. Edge(x, y) was relaxed at that time so $y.d = \phi(s, y)$ when u is added to S.
Now we can obtain a contradiction to prove that $u.d = \phi(s, u)$. Because y appears before u on a shortest path from s to u and all edge weights are non-negative(which means p2 is non-negative), we have $\phi(s, y) \leq \phi(s, u)$, and then

$$y.d = \phi(s, y) \leq \phi(s, u) \leq u.d$$

The statement $\phi(s, u) \leq u.d$ is because $\phi(s, u)$ is the lowest value that $u.d$ can get. But because both vertices u and y were in $V - S$ when u was chosen, we have $u.d \leq y.d$.
Combining these 2 equations, we have $y.d = \phi(s, y) = \phi(s, u) = u.d$, which contradicts with our assumption that $y.d = u.d$. This is a contradiction.
So we have $u.d = \phi(s, u)$ when u is added to S. This equation is valid all time.

Termination: After the last step of iteration, $Q - \emptyset$, which means that $S = V$. Therefore, all vertices are in S. For each $t \in V$, We have found a simple path $P_{s,t} = \{(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, t)\}$ connecting s to t, such that, letting S denote the set of all simple paths connecting s and t, $c(P_{s,t}) = \min_{P \in S} c(P)$. □

Proof of Running Time

Proof. As we know, the running time of Dijkstra algorithm with fibonacci heap is $O(|E| + |V| \log |V|)$. In this case, the only difference between this algorithm and dijkstra is the RELAX part and the time complexity of RELAX part in both algorithm is O(1). So the complexity of this algorithm is the same as Dijkstra, which is $O(|E| + |V| \log |V|)$.
From another perspective, the while loops in lines 4-10 takes $O(|V|)$ for the loop and $O(\log |V|)$ for the heap, so the time complexity for the heap in total is $O(|V| \log |V|)$.
The for loop in lines 7-9 goes through all edges in the graph exactly once, so the total complexity is O(E). So the total time complexity is $O(|E| + |V| \log |V|)$. □

2 Problem 2 (15 points)

You are given a simple weighted graph (no self-edges or multi-edges) $G = (V, E)$ and its minimum spanning tree T_{mst}. Somebody added a new edge e to the graph G. Let's call new graph $G' = (V, E \cup \{e\})$. Devise an algorithm which checks if T_{mst} is also a minimum spanning tree for the graph G' or not. Your algorithm should work in $O(|V|)$ time. Prove correctness of your algorithm and provide running time analysis.

You can assume that all edge weights in G' are distinct. T_{mst} and G' are given to you as adjacency lists.

Solution:
My algorithm tends to first add the new edge to the T_{mst}. As a result, there must be a new loop generated by this operation. After that I will detect that cycle by searching using DFS. After that I gonna check if there is an edge t that has larger weight than e. If so, than deleting edge t will give us a better MST for the new graph than original MST. Thus T_{mst} is is also a minimum spanning tree for the graph G'. If such edge t doesn't exist, then T_{mst} is not a minimum spanning tree for the graph G'.
You can see the pseudo-code below.

Proof of correctness:

Proof. T_{mst} is a tree, so it does not contains any cycle. When you add a new edge e in the graph, there must be a new cycle M appeared. And e must be in M. As we learned in the class, the largest edge of a cycle must not be in the MST(Minimum spanning tree). So the largest edge of M must not be in the new MST. If the largest edge is e, than deleting e will get a minimum spanning tree of the new G', which is exactly original T_{mst}. This means that T_{mst} is also a minimum spanning tree for the graph G'. Otherwise, you must delete the largest edge to get a MST for G', which means that T_{mst} is not a minimum spanning tree for the graph G'.
In this algorithm ,I first go for a DFS to find the cycle. My DFS is very similiar to DFS we taught on class except one thing. I set the vis value of node which we are visiting to -1. So if you find a -1 on the next node you plan to visit, It means that currently your search comes from that node. This means that you are in a cycle. Through this trick, I can find the cycle with one pass of DFS. Then I can simply find all edges in this cycle by backtracking and save these edges in Q.

Algorithm 2 Check_MST (G, e)

```
Q is a array for storing all edges in the loop, Q ← ∅
vis is a array saving the visit status of nodes for DFS, 0 is not visit, -1 is visiting, 1 is visited, vis gets all 0s at the begining.
procedure CHECK_MST(G, e)
    G(V, E) ← (V, E ∪ e)
    DFS(G, e.u)
    return Check(G, Q, e)
end procedure
procedure DFS(G, e.u)
    for each edge t connected to e.u except e do
        if vis[t] == 1 then
            continue
        else if vis[t] == 0 then
            vis[t.u] ← −1
            isLoop = DFS(G, t.u)
            vis[t.u] ← 1
            if isLoop == true then
                Q ← Q ∪ t
                return true
            end if
        else
            Q ← Q ∪ t
            return true
        end if
    end for
end procedure
procedure CHECK(G, Q, e)
    for each edge t in Q do
        if t.weight > e.weight then
            return False
        end if
    end for
    return True
end procedure
```

In procedure Check, since all edges of the cycle is in Q, I can just simply go through the set Q to find if e is the largest edge in this cycle. If the largest edge is e, this means that T_{mst} is also a minimum spanning tree for the graph G'. Otherwise, it means that T_{mst} is not a minimum spanning tree for the graph G'. □

Proof of running time:

Proof. For the procedure Check_MST, line 4 takes O(1) time. Line 5 depends on procedure DFS. In procedure DFS, We search through the whole map. Because we use the vis array to mark all the nodes we visited, so every node will only be searched once. Thus, the time complexity for DFS is O(n). Line 5 depends on procedure Check. In Check, since we only go through all edges in Q, and the number of edges in Q in O(n) because it has at most n edges, the same number as edges in G. So the time complexity of Check is O(n). Thus, the time complexity of Check_MST is O(n). Therefore, the total time complexity is O(n). □

3 Problem 3 (20 points)

An airline has n flights. 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".

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". Your algorithm should run in $O(|L|^2 m)$ time.

Hint: You should just need to show how this can be reduced to a network flow problem and then use a blackbox algorithm solving the flow problem. Prove that your reduction is correct and that the runtime of the algorithm you use will be $O(|L|^2 m)$ for the reduction you give.

Solution: This problem can be seen as a bipartite graph one-to-many matching problem. On the left part of this bipartite is all passengers and on the right part of this bipartite is all flights.
The solution to the problem is to transform the matching problem to max flow problem and use network flow to solve the problem. The steps are as follow:
1.create a bipartite graph. On the left we set one node for each passenger. On the right we set one node for each flights. The edges are set due to the passenger's requirement. Set an edge between each passenger and each flight in his requirement of flights. The capacity of all edges here is 1.
2.Let's create a source with an edge of capacity K_i to each node r_i on the left, and create a sink with edges of capacity 1 from all flights to the sink.
3.After we set the network as above, the problem has become a max flow problem. We can simply use the Ford-Fulkerson algorithm to calculate the max flow within $O(|L|^2 m)$ time.
4.Check whether the max flow k is equal to $\Sigma_i k_i$. If k is equal to $\Sigma_i k_i$, then the passenger will not experience "re-accommodation", otherwise the passenger will experience a "re-accommodation".

Proof of correctness:

Proof. The problem is very similar to the problem we talked on class - the bipartite graph matching graph. Here I gonna prove that the passenger can avoid "re-accommodation" if and only if there is a flow of value k equal to $\Sigma_i k_i$.
First, if the passenger avoid the "re-accommodation", then max flow of value k must exists. This is because that for taking each flight we have a flow of 1. If all the requirements are satisfied, then they must take $\Sigma_i k_i = k$ number of flights. Since each flight must be taken at most once, each taken flight will send 1 unit of flow to the sink. So the flow of value k must exits.True.
In the other direction, if there exists a max flow k, the outcast of the source only have total $\Sigma_i k_i = k$ capacity, then the edges from the source to the flights must be full with 1 unit flow. Beside, each flight can only have 1 unit of flow going through, representing the passenger who takes this flight. Therefore, each passenger's requirement must be satisfied and no passenger get "re-accommodation". True.
Since the passenger can avoid "re-accommodation" if and only if there is a flow of value k equal to $\Sigma_i k_i$, what we need to do is to check whether having a flow of value k is possible. This is because k is the maximum capacity of this network owing to the sum of outcast of source. This is very simple by running the max flow algorithm. □

Proof of running time:

Proof. 1. For reading the input, there are total n(which is $|L|$) flights and m passengers. For each passenger, there are at most L flights in his requirement. So the total number of edges between passengers and flights are $O(|L|m)$. Therefore, there are total m passengers and $O(|L|)$ flights and $O(|L|m)$ links between them , so the reading time is $O(|L|m)$.
2.constructing the flow network. During the constructing of the network, we can find out that there are 1 source, m links from source to passengers, m passengers, $O(|L|m)$ links from passengers to flights, $O(|L|)$ flights, $O(|L|)$ links from flights to sink and one sink. So constructing the whole network only takes $O(|L|m)$ time.
3.running a max-flow algorithm on the network. As we have known in the class, Ford-Fulkerson takes O(OPT(V + E)) time. In this problem, since all links connected to the sink are from flights and have capacity 1, the OPT is equal to $|L|$. In this network,$V = O(1+m+|L|+1) = O(m+|L|)$, $E = O(m+|L|m+|L|) = O(|L|m)$, so the total running time is $O(|L| * (O(m + |L|) + O(|L|m))) = O(|L|^2 m)$.

4.outputting your answer based on the answer of the max-flow algorithm. In the answer part each passenger can only take at most $|L|$ flight, so the total time complexity of this part is $O(m|L|)$
In summary, the time complexity is $O(|L|^2 m)$ $\square$