Homework #7
Introduction to Algorithms/Algorithms 1
600.463
Spring 2016

Submitted by: Sindhuula Selvaraju
JHED ID: sselvar4

April 4, 2016

1 Problem 1 (20 points)

You are given a graph $G = (V, E)$ which represents a computer network. For each edge $e_i \in E$ its weight $w(e_i)$ represents the probability of failure of this edge $w(e_i) = p_i$, where $0 < p_i < 1$. All failures happen independently, thus the probability that there will be a failure on the path $e_{i_1} \to e_{i_2} \to e_{i_3}$ is $1 - (1 - p_{i_1})(1 - p_{i_2})(1 - p_{i_3})$. You want to develop an algorithm which finds NOT the shortest path but the path with the lowest probability of failure. Prove correctness and provide running time analysis. Full score will be given for algorithm working in $O(|V|^2)$ time.

1.1 Algorithm:

The general idea of the algorithm is to minimize $1 - (1 - p_{i_1})(1 - p_{i_2})(1 - p_{i_3})$ i.e. the probability of failure to do this we maximize $(1 - p_{i_1})(1 - p_{i_2})....(1 - p_{i_n})$.
For this we can use a slightly different version of Dijkstra's algorithm.
Initialize-Single-Source(G,s) refers to the function on pg 648 of textbook
ExtractMax(Q) pops the maximum value from the queue

Algorithm 1: Finding pah with lowest probability of failure

Input : Graph G, weight array w, starting point s

Output: Path with lowest probability of failure

```
function MyDijkstra $(G, w, s)$
{
        Initialize-Single-Source(G,s)
        S = $\emptyset$
        Q = G.V
        while Q $\neq \emptyset$
        max = ExtractMax(Q)
        S = S $\cup$ max
        Foreach v in G.Adj[u]
                MyRelax(u,v,1-w)
}
function MyRelax $(u, v, w)$
{
        if($v.d < u.d + w(u, v)$)
            $v.d = u.d + w(u, v)$
            $v.\pi = u$
}
```

1.2 Proof of Correctness:

Since we are trying to minimize $1-(1-p_{i_1})(1-p_{i_2})(1-p_{i_3})$ i.e. the probability of failure and to do this we maximize $(1-p_{i_1})(1-p_{i_2})....(1-p_{i_n})$. To do this we pass 1-w to the MyRelax function. This function updates v.d if a higher value is obtained for a given v using a different v.π
We extract the max value from the queue ensuring that values added to the Set S are maximum values
These changes only affect the weight updation metrics for Djkistra's algorithm and not the behaviour of the algorithm itself. Each new edge added to S is the maximum possible value for reaching a vertex from the edges that have already been analysed.
Since the overall algorithm is not modified we can still assume the soundness of Dijkstra's algorithm.
Hence our algorithm is correct.

1.3 Running Time Analysis:

This algorithm only makes minor modifications to Dijkstra's algorithm like changing the comparison operator and selecting the maximum element instead of the minimum.
These changes only affect the weight metrics and not the complexity of Dijkstra's algorithm..
ExtractMax also takes a linear time just like Extract-Min in traditional Dijkstra, its time complexity is O(V).
Total Time Complexity = Time taken to initiatize the array + Time taken by ExtractMax (which will be called V times in the worst case) + Time taken for V calls of MyRelax
= O(V) + O(V^2) + O(V^2)
= O(V^2)
Hence Proved

2 Problem 2 (20 points)

You are given a graph $G = (V, E)$. The graph is connected and has at most $|V| + 10$ edges. Provide an algorithm with running time $O(|V|)$, which finds the minimum spanning tree of G. You may assume all edges have distinct weights. Prove correctness and provide running time analysis.

2.1 Algorithm:

The main idea behind the algorithm is to apply BFS.
Running BFS on G will produce a spanning tree T with V-1 edges.
For each edge e = (u,v) in G.E try placing it in T
If it already exists in T then do nothing.
If it is not in T, a cycle will be created.
Run BFS again from u to v and keep track of the edge with the maximum weight while traversing.
Delete the edge with the maximum weight in the cycle.
After performing this procedure for all edges, we shall have the minimum spanning tree of G

2.2 Proof of Correctness:

Since, all edges have unique weights a single unique MST exists.
Using the cycle property proved in Homework 6 we know that the heaviest edge of a cycle cannot be part of the MST.
Since a connected graph has at least V-1 edges and every subsequent edge creates a cycle, in our case 11 cycles will be created for the atmost $|V| + 10$ edges and on deleting the largest edge of each cycle we will have the MST of graph G
HEnce Proved

2.3 Running time analysis:

Time taken by BFS on graph G to create a spanning tree T = O($|V| + |E|$)
We try to place each of the $|V| + 10$ edges in T and all further operations are only done if the edge is not there in T.
As V-1 edges out of the total $|V| + 10$ edges will already be present in the spanning tree we do these operations at most 11 times.
Now, if the edge e = (u,v) is not present in the spanning tree, we run a BFS from u to v.
This again will take O($|V| + |E|$) time.
Total time taken

= O($|V| + |E|$) + 11(O($|V| + |E|$))
=12 O($|V| + |E|$) =12 O($V + |V| + 10$) =12 O($2|V| + 10$)
= O($|V|$)
Hence Proved.

3 Problem 3 (10 points)

You are given an unweighted directed graph $G = (V, E)$ and nodes s and t. All nodes in the graph are colored either green, yellow or white. Provide an algorithm which determines if there is a path (might be not simple path) from s to t which goes through both green and yellow vertices at least once, and if so outputs such a path. Full score will be given for running time $O(|V| + |E|)$. Prove correctness and provide running time analysis.

3.1 Algorithm

We can solve this problem by applying BFS in multiple phases.

1. Run BFS from s and take note of all the green and yellow vertices it connects. Also take note of the path traversed.
2. Define a point s_1 which is the first green vertex found in the BFS, that we connect to all the green vertices obtained from step 1. If s is green then connect it also. Similarly define a point s_2 for all yellow vertices.
3. Reverse the direction of each edge in G to form G'
4. Run BFS in G' from t and take note of all yellow and green vertices that connect to t. Also take note of the corresponding paths
5. Define point t_1 which is the first yellow vertex found by BFS that we connect to all yellow vertices obtained from the previous step. If t is yellow then connect it also. Similarly define a point t_2 to connect to all green vertices(including t if applicable).
6. Run BFS from s_1 to t_1. If the path exists then take note of the green and yellow points and add the path to the path from s to s_1 and t_2 to t. Such a path will definitely have both green and yellow nodes.
7. If this path does not exist then run BFS from s_2 to t_2 If the path exists then take note of the green and yellow points and add the path to the path from s to s_2 and t_2 to t. Such a path will definitely hace both green and yellow nodes.
8. If this path also does not exist then return false.

3.2 Proof of Correctness:

We know from the completeness of BFS that if a path exists from s to t through green and yellow nodes,BFS will return it.
This is also true for the reverse graph(we are not really inverting the graph but instead trying to find path from t ot s instead of s to t)
Since s_1,s_2,t_1 and t_2 connect to all their corresponding points in the green and yellow regions (including s and t if they are green or yellow) it accounts for all situations (like s being yellow, t being white etc.)
Thus if a path exists from s_1 to t_1 then on combining paths from s to s_1 with s_1 to t_1 and then t_1 to t we will have a path with s-green node-yellow node-t with other vertices in between.
Similarly,if a path exists from s_2 to t_2 then on combining paths from s to s_2 with $s - 2$ to t_2 and then t_2 to t we will have a path with s-yellow node-green node-t with other vertices in between.
This algorithm also accounts for the case where the solution is not a simple path.
When we restart BFS from an intermediate point s_1,s_2,t_1 or t_2 it can pass through s and t as they have not yet been encountered in this BFS run.
The algorithm returns false if no such path exists.
Hence Proved

3.3 Running Time Complexity:

We run the BFS algorithm k number of times.
Time complexity of this is $k.O(|V| + |E|)$.
The time taken for reversing a graph is O($|E|$).
Total Time Complexity = k.O($|V| + |E|$) + O($|E|$) = O($|V| + |E|$)
Hence Proved