Hovhannes Tsakanyan

HW 6 (Section A)

1. We can simply implement this optimization of BELLMAN-FORD algorithm by remembering if v was relaxed or not. If v is relaxed then we wait to see if v was updated (which means being relaxed again). If v was not updated, then we would stop.
   Because the greatest number of edges on any shortest path from the source is m, then the path-relaxation property tells us that after m iterations of BELLMAN-FORD, every vertex v has achieved its shortest-path weight in v.d. By the upper-bound property, after m iterations, no d values will ever change. Therefore, no d values will change in the (m+1)st iteration. Because we do not know m in advance, we cannot make the algorithm iterate exactly m times and then terminate. But if we just make the algorithm stop when nothing changes any more, it will stop after m + 1 iterations.

2. Begin by calling a slightly modified version of DFS, where we maintain the attribute v.d at each vertex which gives the weight of the unique simple path from s to v in the DFS tree. However, once v.d is set for the first time we will never modify it. It is easy to update DFS to keep track of this without changing its runtime. At first sight of a back edge (u, v), if $v.d > u.d + w(u, v)$ then we must have a negative-weight cycle because $u.d + w(u, v) - v.d$ represents the weight of the cycle which the back edge completes in the DFS tree. To print out the vertices print v, u, $u.\pi$, $u.\pi.\pi$, and so on until v is reached. This has runtime O(V + E).

3. We will compute the total number of paths by counting the number of paths whose start point is at each vertex v, which will be stored in an attribute v.paths. Assume that initial we have v.paths = 0 for all $v \in V$. Since all vertices adjacent to u occur later in the topological sort and the final vertex has no neighbors, line 4 is well-defined. Topological sort takes O(V + E) and the nested for-loops take O(V + E) so the total runtime is O(V + E).

```
1: topologically sort the vertices of G
2: for each vertex u, taken in reverse topologically sorted order do
3:     for each vertex v ∈ G.Adj[u] do
4:         u.paths = u.paths + 1 + v.paths
5:     end for
6: end for
```

4. It does work correctly to modify the algorithm like that. Once we are at the point of considering the last vertex, we know that it's current d value is at at least as large as the largest of the other vertices. Since none of the edge weights are negative, its d value plus the weight of any edge coming out of it will be at least as large as the d values of all the other vertices. This means that the relaxations that occur will not change any of the d values of any vertices, and so not change their π values.

5. We now view the weight of a path as the reliability of a path, and it is computed by taking the product of the reliabilities of the edges on the path. Our algorithm will be similar to that of DIJKSTRA, and have the same runtime, but we now wish to maximize weight, and RELAX will be done inline by checking products instead of sums, and switching the inequality since we want to maximize reliability. Finally, we track that path from y back to x and print the vertices as we go.

```
1: INITIALIZE-SINGLE-SOURCE(G, x)
2: S = ∅
3: Q = G.V
4: while Q 6= ∅ do
5:         u = EXTRACT-MIN(Q) 6: S = S ∪{u}
7:         for each vertex v ∈ G.Adj[u] do
8:                 if v.d < u.d · r(u, v) then
9:                         v.d = u.d · r(u, v)
10:                        v.π = u
11:                end if
12:        end for
13: end while
14: while y 6= x do
15:         Print y
16:         y = y.π
17: end while
18: Print x
```

6. Whenever RELAX sets for some vertex, it also reduces the vertex's d value. Thus if s: gets set to a non-NIL value, s:d is reduced from its initial value of 0 to a negative number. But s:d is the weight of some path from s to s, which is a cycle including s. Thus, there is a negative-weight cycle.

7. a) Since in G_f edges only go from vertices with smaller index to vertices with greater index, there is no way that we could pick a vertex, and keep increasing it's index, and get back to having the index equal to what we started with. This means that G_f is acyclic. Similarly, there is no way to pick an index, keep decreasing it, and get back to the same vertex index. By these definitions, since G_f only has vertices going from lower indices to higher indices, (v1, . . . , $v_{|V|}$) is a topological ordering of the vertices. Similarly, for , G_b ($v_{|V|}$, . . . , v1) is a topological ordering of the vertices.
   b) Suppose that we are trying to find the shortest path from s to v. Then, list out the vertices of this shortest path v_{k_1} , v_{k_2} , . . . , v_{k_m}. Then, we have that the number of times that the sequence $\{k_i\}_i$ goes from increasing to decreasing or from decreasing to increasing is the number of passes over the edges that are necessary to notice this path. This is because any increasing sequence of vertices will be captured in a pass through E_f and any decreasing sequence will be captured in a pass through E_b. Any sequence of integers of length |V | can only change direction at most |V |/2 times. However, we need to add one more in to account for the case that the source appears later in the ordering of the vertices than v_{k_2} , as it is in a sense initially expecting increasing vertex indices, as it runs through E_f before E_b.
   c) It does not improve the asymptotic runtime of Bellman ford, it just drops the runtime from having a leading coefficient of 1 to a leading coefficient of 1 / 2 . Both in the original and in the modified version, the runtime is O(EV).

8. A negative-weight cycle appears when $w_{i,j}^{m} < 0$ for some m and i. Each time a new power of W is computed, we simply check whether or not this happens, at which point the cycle has length m. The runtime is O(n^4).

9. We can recursively compute the values of $\varphi_{i,j}^{(k)}$ by, letting it be $\varphi_{i,j}^{(k-1)}$ if $d_{i,k}^{(k)} + d_{k,j}^{(k)} \geq d_{i,j}^{(k)}$, and otherwise, let it be k. This works correctly because it perfectly captures whether we decided to use vertex k when we were repeatedly allowing ourselves use of each vertex one at a time. To modify Floyd-Warshall to compute this, we would just need to stick within the innermost for loop, something that computes $\varphi_{i,j}^{(k)}$ by this recursive rule, this would only be a constant amount of work in this innermost for loop, and so would not cause the asymptotic runtime to increase. It is similar to the s table in matrix-chain multiplication because it is computed by a similar recurrence.

   If we already have the n^3 values in $\varphi_{i,j}^{(k)}$ provided, then we can reconstruct the shortest path from i to j because we know that the largest vertex in the path from i to j is $\varphi_{i,j}^{(n)}$, call it a_1. Then, we know that the largest vertex in the path before a_1 will be $\varphi_{i,a_1}^{(a_1-1)}$ and the largest after a_1 will be $\varphi_{a_1,j}^{(a_1-1)}$. By continuing to recurse until we get that the largest element showing up at some point is NIL, we will be able to continue subdividing the path until it is entirely constructed.