Exercise Session 11

Exercise 1.
Consider a weighted directed graph $G = (V, E)$ with nonnegative weight function $w: E \to \mathbb{N}$. Solve the following computational problems assuming you can solve the single-source shortest-paths problem using e.g., Dijkstra's algorithm or the Bellman-Ford algorithm. Analyse the running time of your solutions.

Single-destination shortest-paths problem: Find a shortest path to a given destination vertex t from each vertex $v \in V$.

Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v.

All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v.

Solution 1.
Single-destination shortest-paths problem: It can be solved by solving the single-source shortest-paths problem on the transpose of G, denoted by G^T, with source vertex t. We have already seen that computing G^T takes $O(V + E)$. Thus, using Dijkstra's algorithm[1] we can solve the single-destination shortest-paths problem in $O(V + E) + O(V^2) = O(V^2)$. Whereas, using the Bellman-Ford algorithm, it will take $O(V + E) + O(VE) = O(VE)$.

Single-pair shortest-path problem: It can be solved by simply running a single-source shortest-paths algorithm with u as the source. Then we can print the shortest path from u to v by running Print-Path(G, u, v). Again, using using Dijkstra's algorithm the resulting running time is $O(V^2)$. If instead we use the Bellman-Ford algorithm, the resulting running time is $O(VE)$.

All-pairs shortest-paths problem: It can be solved by running a single-source shortest-paths algorithm $|V|$ times, once for each vertex as the source. Since all edge weights are nonnegative, we can use Dijkstra?s algorithm. Thus the running time is $O(V^3)$. If instead we use the Bellman-Ford algorithm, the resulting running time is $O(V^2E)$.

Exercise 2.
Consider a weighted tree $T = (V, E)$ having weight function $w: E \to \mathbb{R}$. The diameter of T is defined as $\max_{u,v \in V} \delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyse the running time of your algorithm.

Solution 2.
Assume the tree is rooted in s. We can find the diameter of the weighted tree by first solving the single-shortest paths problem from s, then determine the maximal shortest path estimate $v.d$ – which at this point is equal to $\delta(s, v)$ – for v ranging in V. Note that a tree is a directed acyclic graph, therefore we can use Dag-Shortest-Path(T, w, s) for solving the single-shortest paths problem.

Weighted-Diam(T, w)
let s be the root of T
Dag-Shortest-Path(T, w, s)
$diam = 0$
for each $v \in T.V$
    $diam = \max(diam, v.d)$
return $diam$

[1] Here we assume the min-priority queue to be implemented using an array, as described in CLRS Section 24.3.

Running DAG-SHORTEST-PATH(T, w, s) takes $O(|V|+|E|)$ and the for loop in lines 4–5 takeds $O(|V|)$. Hence the running time of WEIGHTED-DIAM(T, w) is $O(|V| + |E|)$.

Exercise 3.
(CLRS 24.5-4) Let $G = (V, E)$ be a weighted, directed graph with source vertex s, and let G be initialised by INITAILIZE-SINGLE-SOURCE(G, s). Prove that if a sequence of relaxation steps sets $s.\pi$ to a non-NIL value, then G contains a negative-weight cycle.

Solution 3.
Whenever RELAX sets π for some vertex, it also reduces the vertex's d value. Thus if $s.\pi$ 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.

Exercise 4.
(CLRS 24.3-3) Consider the pseudocode for Dijkstra's algorithm.

DIJKSTRA(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S = \emptyset$
3 $Q = G.V$
4 **while** $Q \neq \emptyset$
5     $u =$ EXTRACT-MIN(Q)
6     $S = S \cup \{u\}$
7   **for each** $v \in G.Adj[u]$
8   RELAX(u, v, w)

Suppose we change guard of the while loop in line 4 as $|Q| > 1$. This causes the while loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm still correct? Motivate your answer.

Solution 4.
Yes, the algorithm still works. Let u be the leftover vertex that does not get extracted from the priority queue Q. If u is not reachable from s, then by the no-path property (Corollary 24.12) $d[u] = \delta(s, u) = \infty$. If u is reachable from s, there is a shortest path $p = s \rightsquigarrow x \to u$. When the vertex x was extracted, $d[x] = \delta(s, x)$ and then the edge (x, u) was relaxed; thus, by the convergence property (Lemma 24.14) $d[u] = \delta(s, u)$.

★ **Exercise 5.**
(CLRS 24-3) Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $49 \cdot 2 \cdot 0,0107 = 1.0486$ U.S. dollars, thus turning a profit of 4.86 percent. Suppose that we are given n currencies $c_1, c_2, \ldots c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j.

(a) Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that $R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_k, i_1] > 1$. Analyse the running time of your algorithm.

(b) Give an efficient algorithm to print out such a sequence if one exists. Analyse the running time of your algorithm.

Solution 5.

(a) Let $G = (V, E)$ be a weighted graph with $V = \{c_1, \ldots, c_n\}$ and weight function

$$w(c_i, c_j) = -\ln R[i, j]\,. \tag{1}$$

Notice that for a sequence $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ we have that $R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_k, i_1] > 1$ iff $\ln R[i_1, i_2] + \ln R[i_2, i_3] + \cdots + \ln R[i_k, i_1] > 0$ (by applying the logarithm on both sides). Taking the negative logarithm this becomes

$$(-\ln R[i_1, i_2]) + (-\ln R[i_2, i_3]) + \cdots + (-\ln R[i_k, i_1]) < 0 \tag{2}$$

The above inequality holds when $p = \langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} c_{i_1} \rangle$ corresponds to a negative cycle in G. Therefore we can conclude that if we can find a negative cycle in G, then we can conclude there exists an opportunity for currency arbitrage.

In this particular problem we can assume that any two vertices u and v are connected by some edge, i.e., $E = V \times V$. This means that any vertex can reach any other with some path, and in particular any negative cycle can be reached starting from any source.

For this reason, the Bellman-Ford algorithm can be used to easily detect negative weight cycles in $O(VE)$ time stating from a source vertex arbitrarily chosen from V.

The algorithm described above is summarised by the following pseudocode.

```
ARIBITRAGE(R)
  Construct G with weight function w defined as in Eq. (1)
  pick some v ∈ G.V
  return ¬BELLMAN-FORD(G, w, v)
```

The worst-case running time of the above algorithm is $O(V^3)$ because we are calling one instance of the Bellman-Ford algorithm algorithm —which runs in $O(VE)$ time— on a graph with n vertices and n^2 edges.

(b) Recall that BELLMAN-FORD, after relaxing all edged $|V| - 1$ times, all estimates represent the weight of a shortest path from the source in case there are no negative cycles. The last loop checks if there is still opportunity to "shorten" a path (i.e., decrease the shortest path weight estimate) by relaxing another edge. If this is the case for some edge $(u, v) \in E$, then we know, that there is a negative cycle from $u \to v \rightsquigarrow u$. To print such a cycle it suffices to print the the path $v \rightsquigarrow u$ induced by the predecessor graph, that is, call PRINT-PATH(G, v, u).

```
PRINT-ARBITRAGE(R)
1  Construct G with weight function w defined as in Eq. (1)
2  pick some v ∈ G.V
3  if ¬BELLMAN-FORD(G, w, v)
4       determine the edge (u, v) ∈ E such that v.d > u.d + w(u, v)
5       PRINT-PATH(G, v, u)
6  else
7       return FALSE
```

Notice that If a cycle is found it will have at most $|V|$ edges, thus executing line 5 takes $O(n)$ time. Line 4 can be integrated in the BELLMAN-FORD procedure at an additional cost $\Theta(1)$. Therefore, the worst-case running time of PRINT-ARIBITRAGE(R) is $O(n^3)$.