National Taiwan University
Department of Electrical Engineering
Algorithms, Fall 2019

Handout #29
December 27, 2019
TAs: Shang-Chien Lin, Yi-Ting Lin

Sample Solutions to Homework #4

1. (15) Exercise 23.1-11

   If we were to add in this newly decreased edge to the given tree, we would be creating a cycle. Then, if we were to remove any one of the edges along this cycle, we would still have a spanning tree. This means that we look at all the weights along this cycle formed by adding in the decreased edge, and remove the edge in the cycle of maximum weight. This does exactly what we want since we could only possibly want to add in the single decreased edge, and then, from there we change the graph back to a tree in the way that makes its total weight minimized.

2. (20) Exercise 23.2-7

   Let $T = (V, E_T)$ be the given minimum spanning tree of $G = (V, E)$. Let $T' = (V \cup \{v\}, E_{T'})$ be a minimum spanning tree of $G' = (V \cup \{v\}, E \cup E_v)$, where v is the new added vertex to G and $E_v = \{(u, v) : u \in V\}$ is the set of corresponding incident edges. We show that we can update T into T' in $O(V \lg V)$ time as follows.

   First, we claim that $E_{T'} \subseteq (E_T \cup E_v)$; that is, T' contains only the edges in $E_T \cup E_v$. To prove this claim, consider a spanning tree T_s of G', where T_s contains an edge set $E_s \subseteq G - E_T$. For any edge $e_s = (x, y) \in E_s$, we can always find an edge $e_t \in E_T \cup E_v$, where e_t incidents on x or y and e_t's weight is less than or equal to e_s's weight because T is a minimum spanning tree of G. That is, we can always replace all edges in E_s by the corresponding edges in $E_T \cup E_v$ and obtain a spanning tree whose cost is less than or equal to the cost of T_s. Thus, for any minimum spanning tree containing edges in $G - E_T$, we can always find another minimum spanning tree containing only the edges in $E_T \cup E_v$.

   Second, because $E_{T'} \subseteq E_T \cup E_v$ holds true, T' is the subgraph of $G_r = (V \cup \{v\}, E_r) = (V \cup \{v\}, E_T \cup E_v)$. Because $|E_T| = |V| - 1$ and $E_v \leq |V|$, we immediately know that $|E_r| = |E_T \cup E_v| \leq |E_T| + |E_v| < 2|V| = O(V)$. Thus, we can find T' in G_r using a minimum-spanning-tree algorithm (such as Kruskal's algorithm) in $O(E_r \lg V) = O(V \lg V)$ time.

3. (30) Exercise 23.4

   (a) This does return an MST. To see this, we'll show that we never remove an edge which must be part of a minimum spanning tree. If we remove e, then e cannot be a bridge, which means that e lies on a simple cycle of the graph. Since we remove edges in nonincreasing order, the weight of every edge on the cycle must be less than or equal to that of e.

   To implement this, we begin by sorting the edges in $O(E \lg E)$ time. For each edge we need to check whether or not $T - e$ is connected, so we'll need to run a DFS. Each one takes $O(V + E)$, so doing this for all edges takes $O(E(V + E))$. This dominates the running time, so the total time is $O(E^2)$.

   (b) This doesn't return an MST. To see this, let G be the graph on 3 vertices a, b, and c. Let the edges be (a, b), (b, c), and (c, a) with weights 3, 2, and 1 respectively. If the algorithm examines the edges in their order listed, it will take the two heaviest edges instead of the two lightest.

   An efficient implementation will use disjoint sets to keep track of connected components. Trying to union within the same component will create a cycle. Since we make $|V|$ calls to MAKESET and at most $3|E|$ calls to FIND-SET and UNION, the runtime is $O(E\alpha(V))$.

   (c) This does return an MST. To see this, we know that the only edges we remove are the edges of maximum weight on some cycle, and there always exists a minimum spanning tree which doesn't include these edges. Moreover, if we remove an edge from every cycle then the resulting graph cannot have any cycles, so it must be a tree.

To implement this, we use the approach taken in part (b), except now we also need to find the maximum weight edge on a cycle. For each edge which introduces a cycle we can perform a DFS to find the cycle and max weight edge. Since the tree at that time has at most one cycle, it has at most $|V|$ edges, so we can run DFS in $O(V)$. The runtime is thus $O(EV)$.

4. (30) Exercise 24.1-6

The Bellman-Ford algorithm initiates single source and runs $|V|-1$ passes of relaxing all edges (in some order). Instead set all vertices' distance to 0 and run $|V|$ passes of relaxing all edges, as follows.

Now, we obtain the graph G_π defined by the predecessor edges $(v, v.\pi)$, for all vertices v for which $v.d$ finite. Run DFS on G_π. If G_π is detected as acyclic, then s has no vertex reachable via a negative cycle. Otherwise, if G_π has a cycle, then that cycle is a negative weight cycle.

```
Modified-Bellman-Ford($G, w$)
1  $v.d \leftarrow 0$ for all $v \in V$
1  $v.\pi \leftarrow$ NIL for all $v \in V$
2  **repeat** $|V|$ times **do**
3      **for** each edge $(u, v) \in G.E$ **do**
4          **if** $v.d > u.d + w(u, v)$ **then**
5              $v.d = u.d + w(u, v)$
6              $v.\pi = u$
```

```
Find-Negative-Cycle($G_\pi$)
1  Perform DFS on $G_\pi$
2      **if** there is no backedge found **then**
3          **return** NIL
4      **else** the backedge (v, u) together with stack vertices
           from $u$ till $v$ forms a negative wt. cycle
5          **return** the vertices of this cycle
```

Figure 1: The Pseudo-code for listing one negative-weight cycle.

Time complexity analysis:

Modified-Bellman-Ford takes time $O(|V||E|)$ as the Bellman-Ford algorithm; Find-Negative-Cycle runs DFS algorithm and needs time bounded by $O(|V|+|E|)$. Therefore, the overall time of the algorithm is bounded by $O(|V||E|)$, which is the same as that of the Bellman-Ford algorithm.

Proof:

Suppose G_π has a cycle. Then, this cycle is a negative weight cycle.

Let $(v_0, v_1, v_2, ..., v_k = v_0)$ be a cycle C in G_π. Say that the relaxation of edge (u, v) causes an update if the condition $v.d > u.d + w(u, v)$ is true and this triggers the relaxation update: $v.d = u.d + w(u.v); v.\pi = u$.

In the sequence of relaxations executed by the algorithm, let the edge (v_{k-1}, v_k) be the last edge whose relaxation caused an update among all the edges of the cycle C. Note that this can be assumed without loss of generality (by renumbering the indices of the vertices of the cycle). Let t_j be the last instant at which the relaxation of the edge (v_{j-1}, v_j) caused an update, for $j = 1, 2, ..., k$. By assumption, $t_k > t_j$ for $j = 1, 2, ..., k-1$. Then, just after this instant t_j, when the relaxation update step is completed, we have, $v_j.d = v_{j-1}.d + w(v_{j-1}, v_j)$. At all time instants subsequent to t_j, $v_j.d$ does not change. However, it may be the case that $v_j - 1.d$ changes after t_j(that is, $t_{j-1} > t_j$), and if so, $v_{j-1}.d$ can only reduce. Hence, at the instant just before t_k, we have,

$$v_j.d \geq v_{j-1}.d + w(v_{j-1}, v_j),\ j = 1, 2, ..., k-1$$

Now, consider the instant just before the instant t_k. At this time,

$$v_k.d > v_{k-1}.d + w(v_{k-1}, v_k)$$

and the relaxation step is yet to be updated. Then, summing the inequations, we have,

$$\sum_{j=1}^{k} v_j.d > \sum_{j=1}^{k} v_{j-1}.d + \sum_{j=1}^{k} w(v_{j-1}, v_j)$$

Since, $\sum_{j=1}^{k} v_j.d = \sum_{j=1}^{k} v_{j-1}.d$, we have,

$$0 > \sum_{j=1}^{k} w(v_{j-1}, v_j)$$

For more detailed proof, please click on the following link:

`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.1981&rep=rep1&type=pdf`

5. (20) Exercise 24.2-4

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 $O(V + E)$.

```
CountPaths(G)
1   topologically sort the vertices of G
2   for each vertex u, taken in topologically sorted order do
3       for each v ∈ Adj[u] do
4           v.paths = u.paths + 1 + v.paths
5   return the sum of all paths attributes
```

Figure 2: The Pseudo-code for counting the total number of paths.

6. (15) Exercise 24.4-1

See Figure 3. One feasible solution is $\{x_1, x_2, x_3, x_4, x_5, x_6\} = \{v_1, v_2, v_3, v_4, v_5, v_6\} = \{-5, -3, 0, -1, -6, -8\}$.

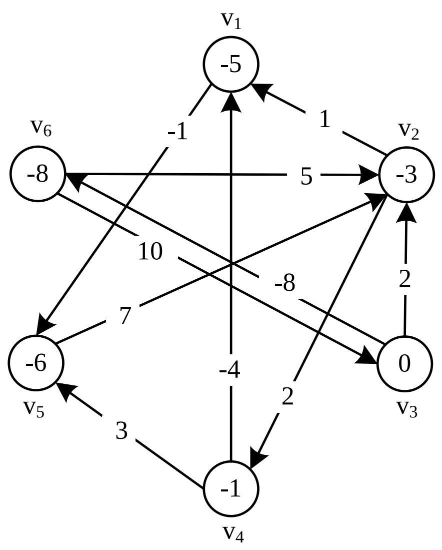

Figure 3: Resulting graph for problem 5.

7. (25) Exercise 24.3

(a) We can use the Bellman-Ford algorithm on a suitable weighted, directed graph $G = (V, E)$, which we form as follows. There is one vertex in V for each currency, and for each pair of currencies c_i and c_j, there are directed edges (v_i, v_j) and (v_j, v_i). (Thus, $|V| = n$ and $|E| = \binom{n}{2}$.)

To determine edge weights, we start by observing that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

if and only if

$$\frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_{k-1}, i_k]} \cdot \frac{1}{R[i_k, i_1]} < 1.$$

Taking logs of both sides of the inequality above, we express this condition as

$$\lg \frac{1}{R[i_1, i_2]} + \lg \frac{1}{R[i_2, i_3]} + \cdots + \lg \frac{1}{R[i_{k-1}, i_k]} + \lg \frac{1}{R[i_k, i_1]} < 0.$$

Therefore, if we define the weight of edge (v_i, v_j) as

$$w(v_i, v_j) = \lg \frac{1}{R[i, j]} = -\lg R[i, j],$$

then we want to find whether there exists a negative-weight cycle in G with these edge weights.

We can determine whether there exists a negative-weight cycle in G by adding an extra vertex v_0 with 0-weight edges (v_0, v_i) for all $v_i \in V$, running BELLMAN-FORD from v_0, and using the boolean result of BELLMAN-FORD (which is TRUE if there are no negative-weight cycles and FALSE if there is a negative-weight cycle) to guide our answer. That is, we invert the boolean result of BELLMAN-FORD.

This method works because adding the new vertex v_0 with 0-weight edges from v_0 to all other vertices cannot introduce any new cycles, yet it ensures that all negative-weight cycles are reachable from v_0.

It takes $\Theta(n^2)$ time to create G, which has $\Theta(n^2)$ edges. Then it takes $O(n^3)$ time to run BELLMAN-FORD. Thus, the total time is $O(n^3)$.

Another way to determine whether a negative-weight cycle exists is to create G and, without adding v_0 and its incident edges, run either of the all-pairs shortest paths algorithms. If the resulting shortest-path distance matrix has any negative values on the diagonal, then there is a negative-weight cycle.

(b) Assuming that we ran BELLMAN-FORD to solve part (a), we only need to find the vertices of a negative-weight cycle. We can do so as follows. First, relax all the edges once more. Since there is a negative-weight cycle, the d value of some vertex u will change. We just need to repeatedly follow the π values until we get back to u. In other words, we can use the recursive method given by the PRINT-PATH procedure of Section 22.2, but stop it when it returns to vertex u.

The running time is $O(n^3)$ to run BELLMAN-FORD, plus $O(n)$ to print the vertices of the cycle, for a total of $O(n^3)$ time.

8. (15) Exercise 25.2-1

The matrices $D^{(k)}$ are

$$D^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}, \quad D^{(1)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix},$$

$$D^{(2)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}, \quad D^{(3)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix},$$

$$D^{(4)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}, \quad D^{(5)} = \begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix},$$

$$D^{(6)} = \begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}.$$

9. (10) Exercise 25.3-4

   It changes shortest paths. Consider the following graph. $V = \{s, x, y, z\}$, and there are 4 edges: $w(s, x) = 2$, $w(x, y) = 2$, $w(s, y) = 5$, and $w(s, z) = -10$. So we would add 10 to every weight to make $\hat{w}$. With w, the shortest path from s to y is $s \to x \to y$, with weight 4. With $\hat{w}$, the shortest path from s to y is $s \to y$, with weight 15. (The path $s \to x \to y$ has weight 24.) The problem is that by just adding the same amount to every edge, you penalize paths with more edges, even if their weights are low.

10. (20) DIY.