National Taiwan University
Department of Electrical Engineering
Algorithms, Fall 2015

Handout #22
January 7, 2016
TAs: Chau-Chin Huang & I-Peng Wu

Sample Solutions to Homework #4

1. (10) Exercise 23.2-8 (pages 637–638)

   This algorithm fails. The corresponding graph is shown in Figure 1.

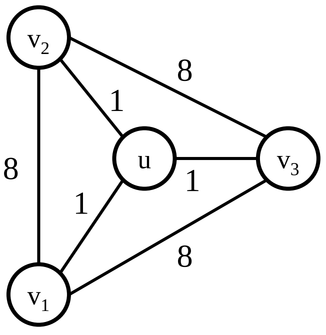

Figure 1: The graph for Problem 1.

   Never mind how the algorithm first divides the graph, the MST in one subgraph (the one containing u) will be one edge of cost 1, and in the other subgraph, it will be one edge of cost 8. Adding another edge of cost 1 will give a tree of cost 10. But clearly, it would be better to connect everyone to u via edges of cost 1.

2. (15) Problem 23-3 (page 640)

   (a) Assume a minimum spanning tree T which is not a bottleneck spanning tree T_b. The maximum weight w of edge e in T must be larger than the maximum weight w_b of edge e_b in T_b. In other words, w is larger than any edge weights in T_b. Thus, we could replace edge e by an edge in T_b and T remains a spanning tree. Then the total edge weight of T would be smaller than original T. Thus, T is impossible to be a minimum spanning tree. In conclusion, a minimum spanning tree T must be a bottleneck spanning tree.

   (b) To determines whether the value of the bottleneck spanning tree is at most b. We can apply DFS once on a random vertex of G ignoring edge weight is bigger than b to see if every edge of G is visited. If is, then the value of the bottleneck spanning tree is at most b.

   (c) We could use an algorithm like binary search to find the value of bottleneck spanning tree. Let b represent the value of the bottleneck spanning tree. We first find the median weight w_m of every edge. Then remove all the edges with weights larger than w_m and use the CHECKBOTTLENECK procedure to determine if $w_m < b$. If it does, we recursively find w_m in the remain edges and do the same procedure. Otherwise, we contract (in textbook Section B.4) all the edge visited in the DFS procedure of the CHECKBOTTLENECK procedure. Then, recursively find w_m in the removed edges in this iteration and do the same procedure. The bottleneck spanning tree T and its value b can be found when there is only one edge in the contracted graph. Since we either contract all the visited edges or decrease half of the remaining edges, the time complexity of this algorithm can be ensured as linear time.

```
CHECKBOTTLENECK($G, b$)
1  for each vertex $u \in V[G]$ do
2      $color[u] \leftarrow$ WHITE
3  $u \leftarrow$ randomly chosen vertex in $G$
4  DFS($u, b$)
5  for each vertex $u \in V[G]$ do
6      if $color[u]$ = WHITE then
7          return $false$
8  return $true$
```

Figure 2: The Pseudo-code for the CHECKBOTTLENECK procedure.

```
DFS($u, b$)
1  $color[u] \leftarrow$ GRAY
2  for each $v \in Adj[u]$ do
3      if $color[v]$ = WHITE and $w(u, v) \le b$ then
4          DFS($v, b$)
5  $color[u] \leftarrow$ BLACK
```

Figure 3: The Pseudo-code for the DFS procedure.

3. (10)

   Given a graph $G = (V, E)$ with weights on edges, run algorithms for all-pairs shortest paths (APSP) to compute the shortest paths for each vertex pair. For each vertex $u \in V$, we record the largest distance $d_{u,v}$ (defined as *radial distance*) where $v \in V - \{u\}$. After that, we find the minimum value among all the largest distances of the vertices. The corresponding source node of the edge obtained above is a center of G. The time complexity is the same as the APSP algorithms adopted.

4. (10) Exercise 24.2-3 (page 657)

   We'll give two ways to transform a PERT chart $G = (V, E)$ with weights on vertices to a PERT chart $G' = (V', E')$ with weights on edges. In each way, we'll have that $|V'| \le 2|V|$ and $|E'| \le |V| + |E|$. We can then run on G' the same algorithm to find a longest path through a directed acyclic graph (in Section 24.2 of the textbook).

   In the first way, we transform each vertex $v \in V$ into two vertices v' and v'' in V'. All edges in E that enter v will enter v' in E', and all edges in E that leave v will leave v'' in E'. In other words, if $(u, v) \in E$, then $(u'', v') \in E'$. All such edges have weight 0. We also put edges (v', v'') into E' for all vertices $v \in V$, and these edges are given the weight of the corresponding vertex v in G. Thus, $|V'| = 2|V|$, $|E'| = |V| + |E|$, and the edge weight of each path in G' equals the vertex weight of the corresponding path in G.

   In the second way, we leave vertices in V alone, but we add one new source vertex s to V', so that $V' = V \cup \{s\}$. All edges of E are in E', and E' also includes an edge (s, v) for every vertex $v \in V$ that has in-degree 0 in G. Thus, the only vertex with in-degree 0 in G' is the new source s. The weight of edge $(u, v) \in E'$ is the weight of vertex v in G. In other words, the weight of each entering edge in G' is the weight of the vertex it enters in G. In effect, we have "pushed back" the weight of each vertex onto the edges that enter it. Here, $|V'| = |V| + 1$, $|E'| \le |V| + |E|$ (since no more than $|V|$ vertices have in-degree 0 in G), and again the edge weight of each path in G' equals the vertex weight of the corresponding path in G.

5. (10) Exercise 24.4-1 (page 669)

   See Figure 4. One feasible solution is $\{x_1, x_2, x_3, x_4, x_5, x_6\} = \{v_1, v_2, v_3, v_4, v_5, v_6\} = \{-5, -3, 0, -1, -6, -8\}$.

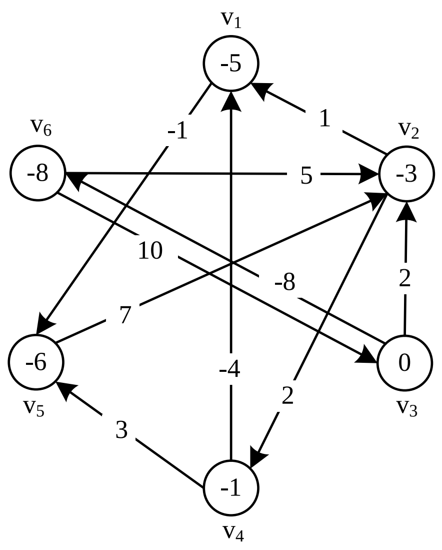

Figure 4: Resulting graph for problem 5.

6. (15) Problem 24-2 (page 678)

   (a) Consider three d-dimensional boxes X, Y, and Z with dimensions $(x_1, x_2, ..., x_d)$, $(y_1, y_2, ..., y_d)$, and $(z_1, z_2, ..., z_d)$, where X nests within Y, and Y nets within Z; that is, there exist permutations π_1 and π_2 such that $x_{\pi_1(i)} < y_i$ and $y_{\pi_2(i)} < z_i$ for $1 \leq i \leq d$, respectively. We can find a permutation π_3 such that $\pi_3(i) = \pi_1(\pi_2(i))$ for $1 \leq i \leq d$ because π_1 and π_2 are valid permutations on d. Thus, we immediately obtain the following results: $x_{\pi_3(i)} = x_{\pi_1(\pi_2(i))} < y_{\pi_2(i)} < z_i$. That is, X nests within Z. The nesting relation is transitive.

   (b) Consider two arbitrary d-dimensional boxes X, Y with dimensions $(x_1, x_2, ..., x_d)$, $(y_1, y_2, ..., y_d)$. We can sort the dimension of X to a non-decreasing array $\langle x_{s_1}, x_{s_2}, ..., x_{s_d} \rangle$ and sort the dimension of Y to a non-decreasing array $\langle y_{t_1}, y_{t_2}, ..., y_{t_d} \rangle$ in $O(d \lg d)$ time. The box X nests inside Y if and only if $x_{s_i} < y_{t_i}$ for $1 \leq i \leq d$. We can check this property in $O(d)$ time. Thus, the method above runs totaly in $O(d \lg d)$ time.

   (c) First, we sort the dimension of every box in the set $\{B_1, B_2, ..., B_n\}$ in $O(nd \lg d)$ time. Second, we build a directed acyclic graph $G = (V, E)$ in $O(n^2 d)$ time, where each vertex $v_i \in V$ corresponds to the box B_i for $1 \leq i \leq n$, and there exists an edge (v_i, v_j) if and only if B_i nests inside B_j, for $i \neq j$, $1 \leq i \leq n$, and $1 \leq j \leq n$. Third, we add a source vertex s and add a directed edge (s, v) to G for each vertex $v \in V$ whose in-degree equals 0. We also add a sink vertex t and add a directed edge v, t to G for each vertex $v \in V$ whose out-degree equals 0. It takes $O(n)$ time to add the two nodes and those edges. Thus, the original longest-sequence-finding problem for nesting boxes can be formulated to a single-pair longest-path problem in G. We can apply a longest-path algorithm from v_s to v_t on G in $O(E+V) = O(n^2)$ time and traverse the longest path to obtain an asked longest sequence of boxes in $O(n)$ time. From the algorithm described above, one can find the solution in $O(n^2 d + nd \lg d)$ time.

7. (10) Problem 24-3 (page 679)

   (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 (699)

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

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

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

9. (10) Exercise 25.3-4 (705)

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. (10)

(a) In Dijkstra's algorithm, use the reliability as the edge weight and substitute.

- max (and EXTRACT-MAX) for min (and EXTRACT-MIN) in relaxation and the queue.
- $\times$ for $+$ in relaxation.
- 1 (identity for $\times$) for 0 (identity for $+$) and $-\infty$ (identity for min) for ∞ (identity for max).

For example, the algorithm in Figure 5 is used instead of the usual RELAX procedure. This algorithm is isomorphic to the one above: It performs the same operations except that it is working with the original probabilities instead of the transformed ones. The time complexity is $O(VlgV+E)$.

```
RELAX-RELIABILITY($u, v, r$)
1  if $d[v] < d[u] \cdot r(u,v)$
2      then $d[v] \leftarrow d[u] \cdot r(u,v)$
3           $\pi[v] \leftarrow u$
```

Figure 5: The Pseudo-code for the RELAX procedure.

(b) To find the maximum reliability path between each pair of vertices, we modify each entry of W matrix as follows:

$$w_{ij} = \begin{cases} 1 & \text{if } i = j \ , \\ \mu_{ij} & \text{if } i \neq j \text{ and } (i,j) \in E \ , \\ 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \ . \end{cases}$$

we can use the algorithm in Figure 6. $d_{ij}^{(k)}$ is the maximum reliability on path $i \rightsquigarrow j$ using vertices with index $\leq$ k. Further, we modify the predecessor matrix Π for $k \leq 1$ as follows:

```
Max-Reliability($W$)
1  $n \leftarrow rows[W]$
2  $D^{(0)} \leftarrow W$
3      for $k \leftarrow 1$ to $n$
4          for $i \leftarrow 1$ to $n$
5              for $j \leftarrow 1$ to $n$
6                  $d_{ij}^{(k)} \leftarrow \max(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} \times d_{kj}^{(k-1)})$
8  return $D^{(n)}$
```

Figure 6: Determine the maximum capacity path between each pair of vertices.

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} \times d_{kj}^{(k-1)} \ , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} \times d_{kj}^{(k-1)} \ . \end{cases}$$

11. (10) Exercise 26.2-3 (page 730)

See Figure 7 for the execution. The maximum flow is 23. According to the Theorem 26.7, since the Edmonds-Karp algorithm uses the framework of Ford-Fulkerson method and ends when there is no augmenting path in the residual network, the resulting flow is maximum.

12. (10) Problem 26-1 (pages 760-761)

(a) See Figure 8. We could model this problem by letting every vertex v with capacity c be an edge e with the same capacity c. All the edges which are directed to v are now connected at the tail of e, and those which are from v are now connected at the head of e. Clearly, if there is a solution in the

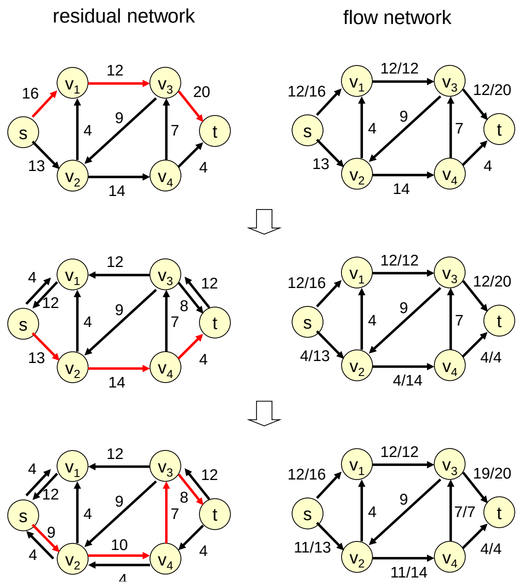

Figure 7: Edmonds-Karp algorithm of problem 26.2-3.

problem with edge and vertex capacities, it could be reduced to a solution in the original max-flow problem. On the other side, a solution in the original max-flow problem can be easily reduced to a solution in the other problem. Thus, we show that a max-flow problem with edge and vertex capacities can be reduced to the original max-flow problem.

(b) We could model this problem into a max-flow problem with edge and vertex capacities. Construct the graph by letting all the starting point be source vertices, all the boundary vertices be the sink vertices, and the other vertices can be general vertices. If any two vertices between source, sink, and general vertices are connected, construct an edge between them. All the capacities of vertices and edges in this graph are 1. Then we could reduce this problem into the original max-flow problem, and apply the Ford-Fulkerson's algorithm to solve it.

Constructing the graph takes $O(V + E)$ time, and reducing the problem also takes $O(V + E)$ time. Ford-Fulkerson's algorithm takes $O(E|f^*|) = O(V \times V) = O(V^2)$ because $O(E) = O(V)$ (for grid graph) and $|f^*| \leq m < V$. Translating the solution back to the original problem takes $O(V + E)$ time. Thus, the time complexity of the whole algorithm is $O(V^2) = O(n^4)$.

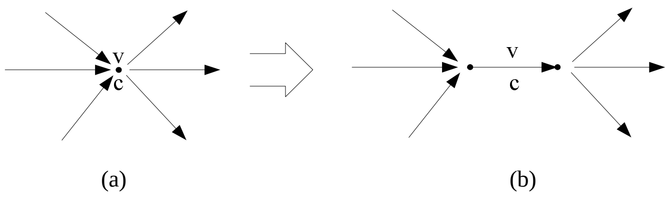

Figure 8: (a) The vertex capacity model. (b) The flow network corresponding to the vertex capacity model.

13. (10)

Let S denotes the source, T denotes the sink, g_i denotes citizen group i, r_i denotes representative i, and c_i denotes committee i. A flow network for the problem instance can be constructed as shown in Figure 9. The maximum flow from the source S to the sink T on this graph is the maximum number of representatives can be selected. By solving the network flow problem using the Ford-Fulkerson algorithm, if the maximum flow $|f^*| = p$, then Ko-P can hold the general committee satisfying the "balancing" property. Otherwise, we have $|f^*| < p$, which indicates the "balancing" property cannot be held. It is obvious that $|f^*| = p = 4$, which indicates the meeting can be held and satisfies the "balancing" property.

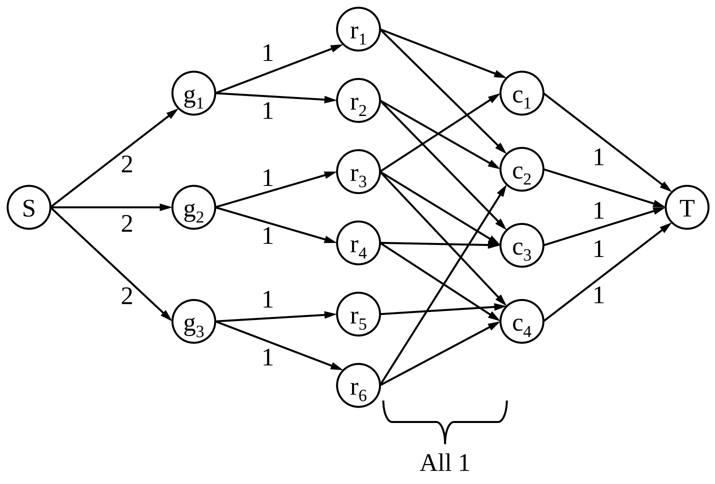

Figure 9: The flow network for Problem 13.

14. (35) DIY.