Homework 7 Solution
CS 581 Spring 2018

Problem 1.

a

1) Always produces a minimum spanning tree, proof:

1. Prove T is a spanning tree.

T is first of all always a connected graph that covers all nodes. For every cycle in T, one edge in the cycle will be removed, since removing an edge from a cycle still keeps the graph connected, which satisfies the condition of removing an edge. Therefore, there can't be any cycle in T, so T is acyclic connected graph that covers all nodes, namely a spanning tree.

2. Prove T is a minimum spanning tree.

Because all minimum spanning trees of a graph have $|V| - 1$ edges, transforming one MST to another MST would require removing k edges and add another k edges($k < |V|$). Every remove/add must happen in a "previous" cycle, i.e. there are more than one choices to remove an edge in a cycle to still remain connected, a remove/add operation is essentially switching the edge removed.

Since the edges are considered in a non-increasing order, it's not possible that in an otherwise cycle an edge with a smaller weight gets removed while the heavier edge remains, which is the only situation that there exist smaller MSTs than T, because the heavier edge will be considered removing first in this algorithm. Therefore, T must be a MST.

2) the most efficient implementation

Let E be the number of edges and V be the number of vertices. The sorting can implemented with any $O(ElogE)$ sorting algorithm.

The loop repeats E times, each time we need to check if vertex u can still be connected with vertex v if edge from u to v is removed. Using a BFS or DFS to check connectivity takes $O(E)$ in the worst case.

The overall time complexity is $O(ElogE + E * E) = O(E * E)$. It computes a MST.

b

1) Sometimes produces a MST, proof:

1. Prove T is a spanning tree.

All nodes will be connected in T because all original edges are considered and only those "redundant" edges are removed. T also has no cycles since it explicitly checks cycle each time.

2. Prove T is not necessarily a minimum spanning tree.

Edges are selected arbitrarily, so it possible that in an otherwise cycle with n edges, the

smallest edge is not included while the other n-1 edges remain. In this case, we can transform the tree into a smaller MST through removing a heavier edge and keeping that smallest edge.

2) the most efficient implementation

Let E be the number of edges and V be the number of vertices. The loop repeats E times. Checking if adding e forms a cycle takes $O(\alpha(V))$ time if using a disjoint set Union-Find data structure.

The overall time complexity is $O(E\alpha(V))$. It computes a MST.

c

1) Always produces a minimum spanning tree, proof:

1. Prove T is a spanning tree.

T is first of all always a connected graph that covers all nodes because every edge that does not cause a cycle is considered.

2. Prove T is a minimum spanning tree.

The reasoning is the similar as a's. Suppose there exists a smaller MST S than T, that means when removing an edge from a cycle, the edge removed for S has a greater weight than the weight removed from T, which is not possible because the algorithm always removes the maximum-weight edge in a cycle.

2) the most efficient implementation

The implementation is similar to b except the extra overhead to find the maximum edge weight in case of a cycle each time, which takes $O(V)$ time. So the overall time complexity is $O(E * V) = O(E * V)$. It computes a MST.

Problem 2.

Run Dijkstra's algorithm on the weighted graph below, using vertex A as the source. Write the vertices in alphabetical order and compute all distances at each step.

Answer:

Step	A->A	A->B	A->C	A->D	A->E	A->F	A->G	A->H
1	0	5	∞	∞	4	∞	∞	∞
2	0	5	∞	∞	4	15	∞	∞
3	0	5	11	∞	4	15	15	∞
4	0	5	11	12	4	15	15	28
5	0	5	11	12	4	15	15	24
6	0	5	11	12	4	15	15	24
7	0	5	11	12	4	15	15	24
8	0	5	11	12	4	15	15	24

Figure 1: Distances calculated by Dijkstra's algorithm at each step.

Problem 3.

a) Give a counter example where Dijkstra's algorithm fails to generate shortest paths in graphs with negative edge weights, but no negative weight cycle. Show your graph, source node and show how Dijkstra's algorithm fails to find the shortest paths.

Answer: In Figure 2 we have a graph with no negative weight cycles, but a negative weight edge. If we have our source node as A then Dijkstra's algorithm will fail to find the shortest path to node D. Since the algorithm terminates when it finds what it believes to be the shortest path it will determine the shortest path from A to D to be $A \to D$ with a weight of 3 when the shortest path is actually $A \to B \to C \to D$ with a weight of 2. This is because the algorithm believes it's done finding the path to D when it compares the weight of $A \to D$ to the cumulative weight of $A \to B \to C$ and finds that $A \to D$ has the lower weight. It will at this point stop looking for paths to D.

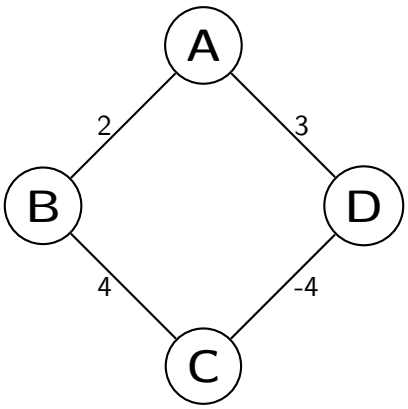

Figure 2: Graph with negative edge weight.

b) Why may Dijkstra's algorithm not find the shortest path when negative-weight edge is present?

Answer: When the algorithm decides it's found the shortest path to a node it will stop looking for other paths to that node. If there are negative edge weights in a path that the algorithm hasn't looked at this could cause the cumulative weight of another path to be lower than the path it's decided is the shortest. As in the example in part a.

c) A man on the street proposes to increment the weight on each edge by a same value s to make all weights non-negative, if there are negative edges, and then run Dijkstra's algorithm. When he finds the shortest paths, he would minus $s * l$ from a path where l is the number of edges on this path to get the final distance. Will this method fix the negative edge problem? Justify your answer.

Answer: No, this method will not fix the negative edge problem. The shortest path may actually have more edges than other paths. This could cause the weight of the shortest path to increase more than the weight of other paths, and would lead to a

similar problem. For a situation where this would happen you can look at the graph in Figure 2 again. Say that we make $s = 5$, which results in Figure 3. In this new graph, we have only made the true shortest path appear even worse and Dijkstra's will still choose $A \to D$.

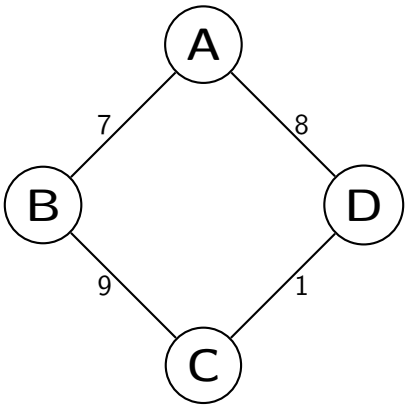

Figure 3: Figure 2 adjusted with $s = 5$.

d) If you believe c) won't work, suggest a method yourself to modify Dijkstra's algorithm so that it works with negative edges (no need to handle negative cycles) and analyze the running time. Your running time should be $O(V^{3-\epsilon})$ where ϵ is a small positive number (polynomially smaller than $O(V^3)$).

Answer: Johnson's algorithm first rewrites all edges weights to eliminate negative edges by introducing a new node and calling a Bellman–Ford algorithm, which has $O(VE)$ running time. Then it calls Dijkstra's algorithm ($O(E + V \log V)$). This is polynomially smaller than $O(V^3)$ when the matrix is sparse ($E \sim V$).

Problem 4.

Another way to reconstruct shortest paths in the Floyd-Warshall algorithm uses values $\phi_{ij}^{(k)}$ for $i, j, k = 1, 2, ..., n$ where ϕ_{ij} is the highest-numbered intermediate vertex of a shortest path from i to j in which all intermediate vertices are in the set $1, 2, ..., k$. Give a recursive formulation for $\phi_{ij}^{(k)}$, modify the FLOYD-WARSHALL procedure to compute the $\phi_{ij}^{(k)}$ values, and rewrite the PRINT-ALL-PAIRS-SHORTEST-PATH procedure to take the matrix $\Phi = \phi_{ij}^{(k)}$ as an input.

Answer: The recursive formula for calculating $\phi_{ij}^{(k)}$ is shown below along with the two modified algorithms.

$$\phi_{ij}^{(k)} = \begin{cases} null & k = 0 \\ k & k > 0, \text{ and } d_{ik}^{(k-1)} + d_{kj}^{(k-1)} < d_{ij}^{(k-1)} \\ \phi_{ij}^{(k-1)} & k > 0, \text{ and } d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \geq d_{ij}^{(k-1)} \end{cases}$$

```
1  MODIFIED-FLOYD-WARSHALL(W)
2      n = W.rows
3      D^(0) = W
4      Φ^(0) = null
5      for k = 1 to n
6          let D^(k) = (d_ij^(k)) be a new n x n matrix
7          let Φ^(k) = (φ_ij^(k)) be a new n x n matrix
8          for i = 1 to n
9              for j = 1 to n
10                 if d_ik^(k-1) + d_kj^(k-1) < d_ij^(k-1)
11                     d_ij^(k) = d_ik^(k-1) + d_kj^(k-1)
12                     φ_ij^(k) = k
13                 else
14                     d_ij^(k) = d_ij^(k-1)
15                     φ_ij^(k) = φ_ij^(k-1)
```

```
1  MODIFIED-PRINT-ALL-PAIRS-SHORTEST-PATH(Φ, i, j)
2      if i == j
3          print i
4      else if Φ_ij == null
5          print "no path from" i " to " j " exists"
6      else
7          MODIFIED-PRINT-ALL-PAIRS-SHORTEST-PATH(Φ, i, φ_ij)
```

Problem 5.

Use the Ford-Fulkerson algorithm to maximize flow on the graph shown below. Specify the algorithm of your choice to find augmenting paths. At each iteration, you should (1) draw the residual graph, (2) find an augmenting path in that graph, and (3) draw the new flow graph resulting from the augmenting path.

Answer: At each step I choose the shortest augmenting path with the largest bottleneck. So for the first path I would choose the bottom path $s \to 4 \to 5 \to 5 \to t$. This path could be found using a BFS similar to Edmond-Karps algorithm.

Path chosen is $s \to d \to e \to t$ with a bottleneck of 4.

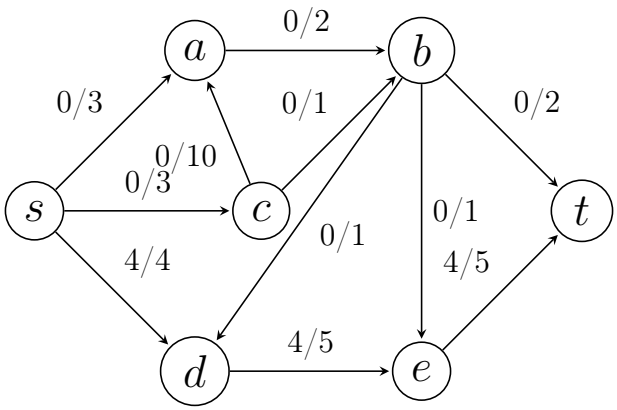

Figure 4: New Flow Graph

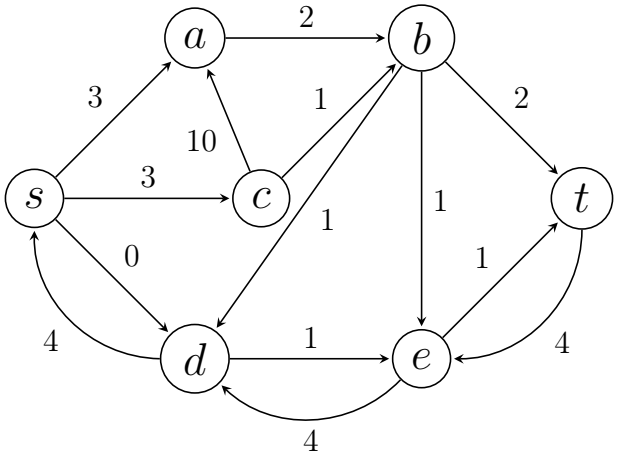

Figure 5: Residual Graph

Path chosen is $s \to a \to b \to t$ with a bottleneck of 2.

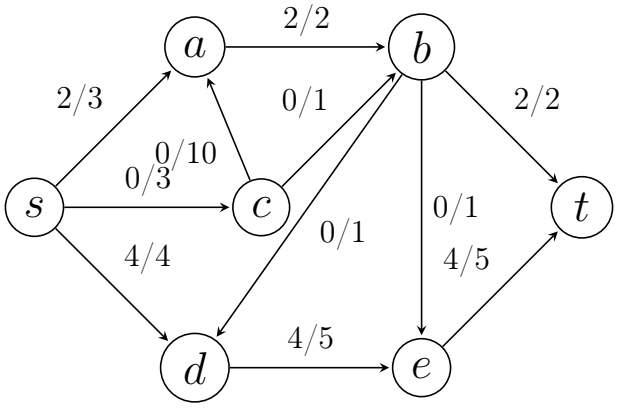

Figure 6: New Flow Graph

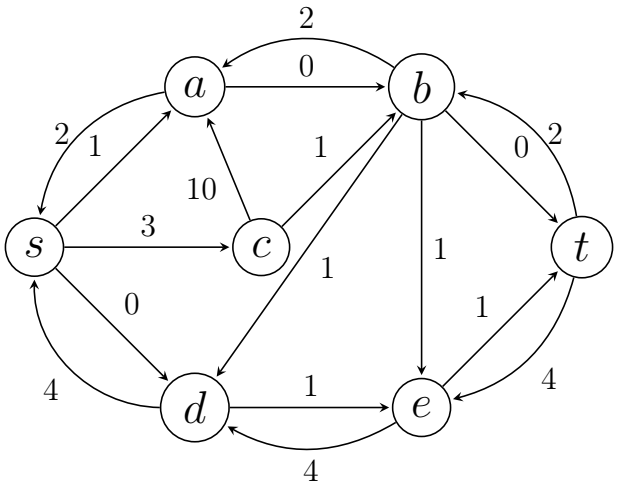

Figure 7: Residual Graph

Path chosen is $s \to c \to b \to e \to t$ with a bottleneck of 1.

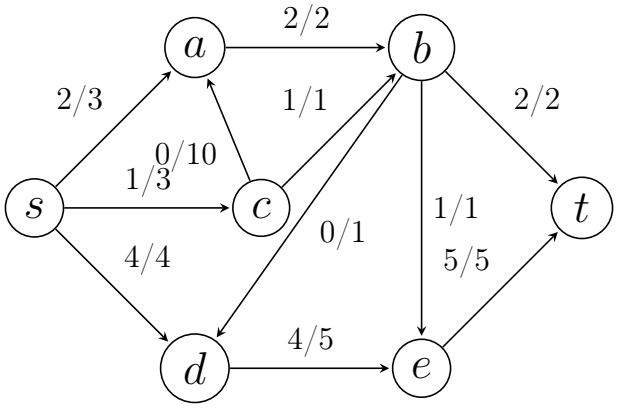

Figure 8: New Flow Graph

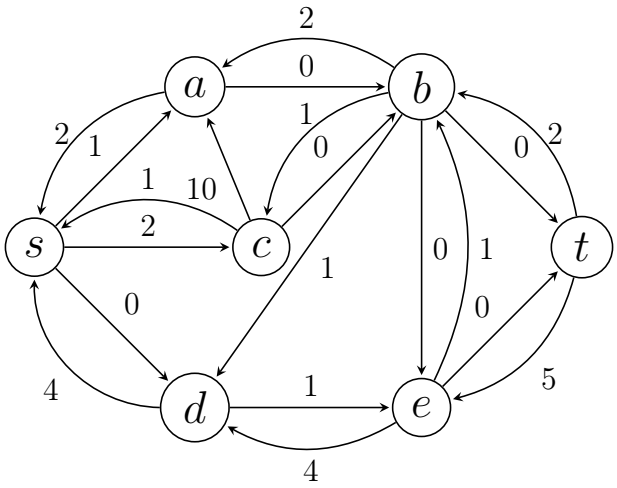

Figure 9: Residual Graph

At this point there are no more augmenting paths and the max flow of the network is known to be 7.

Problem 6.

For vector $r = \langle 1, 2, 3 \rangle$, $c = \langle 2, 1, 2, 1 \rangle$,

a) Find a binary matrix for which the sum of the ith row is r_i and the sum of the jth column is c_j , or prove that such a matrix does not exist.

Answer: To convert the problem to a maximum flow problem, we first add a source node s and a destination node t. Then we add three nodes $r1, r2, r3$ and add edges from s to them. The weights are vector r. Similarly, we add four nodes $c1, c2, c3, c4$ and add edges from them to t. The weights are vector c. Finally add an edge with weight 1 from each $r1, r2, r3$ to $c1, c2, c3, c4$, which are 12 in total and correspond to the 12 cells in the matrix.

After the transformation, we initialize the matrix M to all zeros and run Ford-Fulkerson algorithm. For each augmenting path, if the edge from r_i and c_j is included in the path, then $M[i][j] = 1$. The Ford-Fulkerson is omitted here and the following matrix is a possible result.

$$M = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

b) If the matrix does exist, determine whether it is unique.

Answer: When running Ford-Fulkerson's algorithm, the discovered augmenting paths could be different depending on how you find an augmenting path. So M is not unique.

There are many matrices which would work for vectors r and c. The following matrix is a possible answer.

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$