Computer Algorithms – Homework #4 Solutions

Problem #1 (#22.2-8):

(Solution 1)

We run the BFS twice, the first BFS with any root finding the most distant leaf u from it (the vertex u is discovered very last), and the second BFS with the node u as the root finding the most distant vertex v from it (the vertex v is discovered very last), then the corresponding distance between v and u is the diameter of the tree. The running time is obviously $O(V + E) = O(V)$, since, for trees, $|E| = |V| - 1$.

(Solution 2)

Let v be an arbitrary vertex in G, which we designate as the *root* of the tree. If we remove v from the tree, we obtain several smaller trees whose roots are the vertices that were adjacent to v. We can repeat this process, each time obtaining new roots and smaller trees, until all edges are removed. We will solve the problem by induction on the number of times we have to perform this process until all edges are removed. If the path corresponding to the diameter of the tree does not contain the root, then the diameter of the tree is also the diameter of one of the smaller trees, which we find by induction. If the root is contained in the path corresponding to the diameter, then this path connects two vertices in two separate smaller trees that are farthest away from the root. This observation suggests the following induction hypothesis (which is stronger than the straightforward one).

- **Induction Hypothesis:** *We know how to find the diameter of subtrees with* $< n$ *vertices, and how to find the maximal distance from a fixed root.*

The base case is trivial. Given a tree with n vertices, we designate v as the root, and solve the problem by induction for all the subtrees rooted at the children of v. Notice that the distances we find are those to the children of v, and not those to v. However, to find the distances to v, we need only to add 1 to all the distances. After doing that, we compare the maximum diameter found among the subtrees with the sum of the two maximum distances from v. The larger of the two is the diameter. This is a linear-time algorithm to find the diameter of a given tree.

Problem #2 (#22.4-3):

An undirected graph is acyclic (i.e., a forest) if and only if a DFS yields no back edges.

- If there's a back edge, there's a cycle.
- If there's no back edge, then by Theorem 22.10, there are only tree edges. Hence, the graph is acyclic.

Thus, we can run DFS: if we find a back edge, there's a cycle.

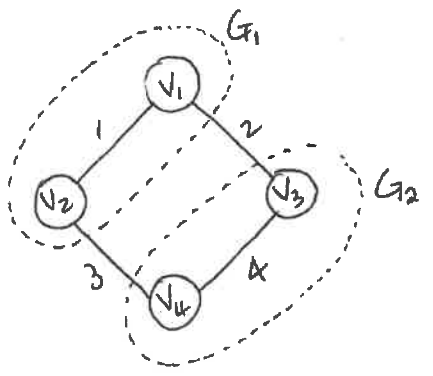
Time Complexity: $O(V)$ (Not $O(V + E)$!)

If we ever see $|V|$ distinct edges, we must have seen a back edge because (by Theorem B.2 on p. 1174) in an acyclic (undirected) forest, $|E| \leq |V| - 1$.

Problem #3 (#23.2-8):

Counterexample to show that the algorithms fails:

Given a graph $G(V, E)$, let us divide the graph into two subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$. Subgraph G_1 consists of the two nodes v_1 and v_2, and subgraph G_2 consists of the other two nodes v_3 and v_4. The minimum-spanning-tree of the subgraph 1 consists of the edge (v_1, v_2) whose weight is 1, and the minimum-spanning-tree of the subgraph 2 consists of the edge (v_3, v_4) whose weight is 4. Following professor Borden's algorithm, let us unit the resulting two minimum spanning trees into a single spanning tree using one of edges in the cut (V_1, V_2), edge (v_2, v_4). Then the solutions of professor Borden's algorithm is the spanning-tree $(v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_3)$ with weight 8, but this is NOT the minimum spanning tree of G.



Problem #4 (#24.1-3):

If 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 every change. Therefore, no d values will change in the $(m + 1)$ iterations. 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.

```
BELLMAN-FORD-$(M + 1)$ $(G, w, s)$

INITIALIZE-SINGLE-SOURCE $(G, s)$
changes ← TRUE
while changes = TRUE do
        changes ← FALSE
        for each edge $(u, v) \in E[G]$ do
                RELAX-M $(u, v, w)$

RELAX-M $(u, v, w)$
if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v), \pi[v] \leftarrow u$, changes ← TRUE
```

The test for a negative-weight cycle (based on there being a d that would change if another relaxation step was done) has been removed above, because this version of the algorithm will never get out of the **while** loop unless all d values stop changing.

Problem #5 (#24.3-8):

Observe that if a shortest-path estimate is not ∞, then it's at most $(|V| - 1)W$. Why? In order to have $d[v] < \infty$, we must have relaxed an edge (u, v) with $d[u] < \infty$. By induction, we can show that if we relax (u, v), then $d[v]$ is at most the number of edges on a path from s to v times the maximum edge weight. Since any acyclic path has at most $|V| - 1$ edges and the maximum edge weight is W, we see that $d[v] \leq (|V| - 1)W$. Note also that $d[v]$ must also be an integer, unless it is ∞.

We also observe that in Dijkstra's algorithm, the values returned by the EXTRACT-MIN calls are monotonically increasing over time. Why? After we do our initial $|V|$ INSERT operations, we never do another. The only other way that a key value can change is by a DECREASE-KEY operation. Since edge weights are nonnegative, when we relax an edge (u, v), we have that $d[u] \leq d[v]$. Since u is the minimum vertex that we just extracted, we know that any other vertex we extract later has a key value that is at least $d[u]$.

When keys are known to be integers in the range 0 to k and the key values extracted are monotonically increasing over time, we can implement a min-priority queue so that any sequence of m INSERT, EXTRACT-MIN, and DECREASE-KEY operations takes $O(m + k)$ time. Here's how. We use an array, say $A[0 \,..\, k]$, where $A[j]$ is a linked list of each element whose key is j. Think of $A[j]$ as a bucket for all elements with key j. We implement each bucket by a circular, doubly linked list with a sentinel, so that we can insert into or delete from each bucket in $O(1)$ time. We perform the min-priority queue operations as follows:

- INSERT: To insert an element with key j , just insert it into the linked list in $A[j]$. Time: $O(1)$ per INSERT.
- EXTRACT-MIN: We maintain an index min of the value of the smallest key extracted. Initially, min is 0. To find the smallest key, look in $A[min]$ and, if this list is nonempty, use any element in it, removing the element from the list and returning it to the caller. Otherwise, we rely on the monotonicity property (and that there is no INCREASE-KEY operation) and increment min until we either find a list $A[min]$ that is nonempty (using any element in $A[min]$ as before) or we run off the end of the array A (in which case the min-priority queue is empty). Since there are at most m INSERT operations, there are at most m elements in the min-priority queue. We increment min at most k times, and we remove and return some element at most m times. Thus, the total time over all EXTRACT-MIN operations is $O(m + k)$.
- DECREASE-KEY: To decrease the key of an element from j to i, first check whether $i \leq j$, flagging an error if not. Otherwise, we remove the element from its list $A[j]$ in $O(1)$ time and insert it into the list $A[i]$ in $O(1)$ time. Time: $O(1)$ per DECREASE-KEY.

To apply this kind of min-priority queue to Dijkstra's algorithm, we need to let $k = (|V| - 1)W$, and we also need a separate list for keys with value ∞. The number of operations m is $O(V + E)$ (since there are $|V|$ INSERT and $|V|$ EXTRACT-MIN operations and at most $|E|$ DECREASE-KEY operations), and so the total time is $O(V + E + VW) = O(VW + E)$.

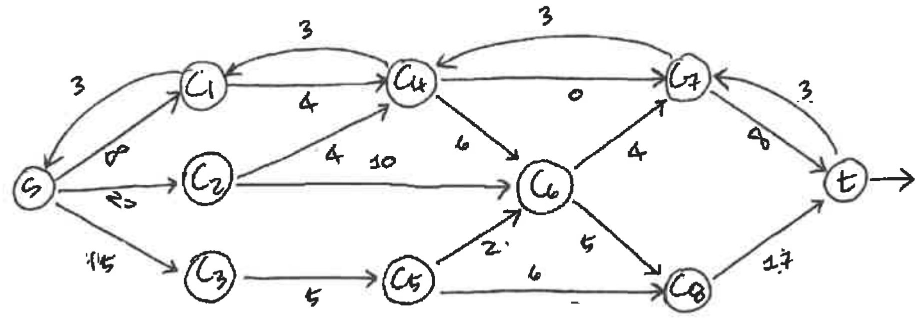
Problem #6

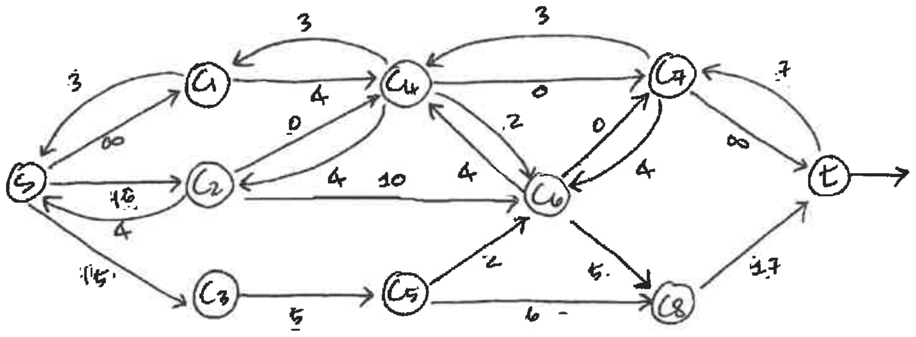
We consider directed graph G= (V,E) which contains negative edges from source s. All other edges are positive.

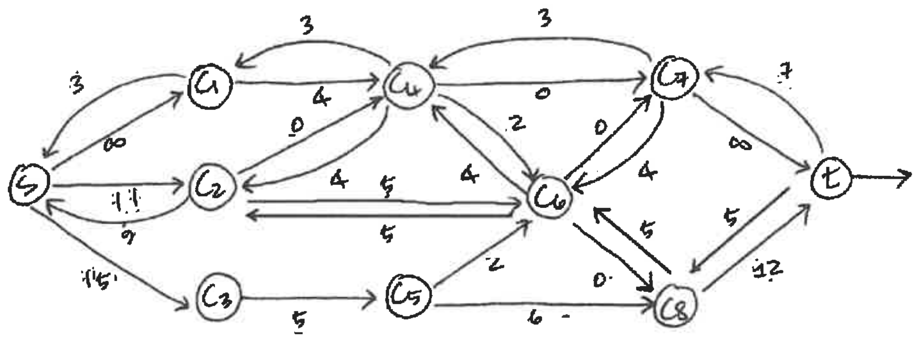
Consider the way apply Dijstra's algorithm to graph G, we can see that all non-source node which connect to negative edges will be added at the beginning of the algorithm since they are always have smaller value than the other. Since all other edges are positive and the graph are directed, the directly way from source s to those nodes are the shortest path from s to them. Hence, when we add those node to set S, $d[v] = \delta(v)$.

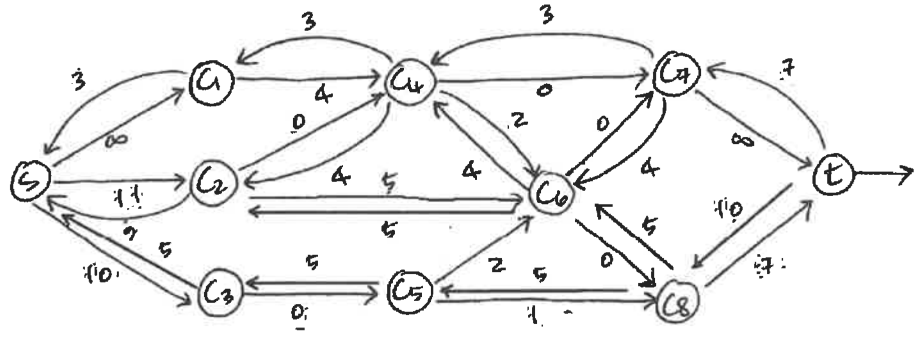
Now, we consider nodes which do not connect to negative edges. Because there is no negative edge in the graph that connect those nodes to each other, we can apply the same arguments of Dijstra's correctness proof to prove the correctness of each time add a node to set S.

Problem #7 (Ford-Fulkerson):









Max flow
= 17 unit flows

Problem #8

$$A^0 = \begin{pmatrix} 0 & 7 & \infty & \infty & \infty & 9 \\ \infty & 0 & \infty & \infty & \infty & 5 \\ \infty & 5 & 0 & \infty & \infty & 4 \\ 2 & \infty & \infty & 0 & 3 & \infty \\ 5 & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 5 & 0 \end{pmatrix} \qquad A^1 = \begin{pmatrix} 0 & 7 & \infty & \infty & \infty & 9 \\ \infty & 0 & \infty & \infty & \infty & 5 \\ \infty & 5 & 0 & \infty & \infty & 4 \\ 2 & 9 & \infty & 0 & 3 & 11 \\ 5 & 12 & \infty & \infty & 0 & 14 \\ \infty & \infty & \infty & \infty & 5 & 0 \end{pmatrix}$$

$$A^2 = \begin{pmatrix} 0 & 7 & \infty & \infty & \infty & 9 \\ \infty & 0 & \infty & \infty & \infty & 5 \\ \infty & 5 & 0 & \infty & \infty & 4 \\ 2 & 9 & \infty & 0 & 3 & 11 \\ 5 & 12 & \infty & \infty & 0 & 14 \\ \infty & \infty & \infty & \infty & 5 & 0 \end{pmatrix} \qquad A^3 = \begin{pmatrix} 0 & 7 & \infty & \infty & \infty & 9 \\ \infty & 0 & \infty & \infty & \infty & 5 \\ \infty & 5 & 0 & \infty & \infty & 4 \\ 2 & 9 & \infty & 0 & 3 & 11 \\ 5 & 12 & \infty & \infty & 0 & 14 \\ \infty & \infty & \infty & \infty & 5 & 0 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 7 & \infty & \infty & \infty & 9 \\ \infty & 0 & \infty & \infty & \infty & 5 \\ \infty & 5 & 0 & \infty & \infty & 4 \\ 2 & 9 & \infty & 0 & 3 & 11 \\ 5 & 12 & \infty & \infty & 0 & 14 \\ \infty & \infty & \infty & \infty & 5 & 0 \end{pmatrix} \qquad A^5 = \begin{pmatrix} 0 & 7 & \infty & \infty & \infty & 9 \\ \infty & 0 & \infty & \infty & \infty & 5 \\ \infty & 5 & 0 & \infty & \infty & 4 \\ 2 & 9 & \infty & 0 & 3 & 11 \\ 5 & 12 & \infty & \infty & 0 & 14 \\ 9 & 17 & \infty & \infty & 5 & 0 \end{pmatrix}$$

$$A^6 = \begin{pmatrix} 0 & 7 & \infty & \infty & 14 & 9 \\ 15 & 0 & \infty & \infty & 10 & 5 \\ 14 & 5 & 0 & \infty & 9 & 4 \\ 2 & 9 & \infty & 0 & 3 & 11 \\ 5 & 12 & \infty & \infty & 0 & 14 \\ 9 & 17 & \infty & \infty & 5 & 0 \end{pmatrix} \qquad \Pi^6 = \begin{pmatrix} N & 1 & N & N & 6 & 1 \\ 6 & N & N & N & 6 & 2 \\ 6 & 3 & N & N & 6 & 3 \\ 4 & 1 & N & N & 4 & 1 \\ 5 & 1 & N & N & N & 1 \\ 5 & 5 & N & N & 6 & N \end{pmatrix}$$