COMP 3804 Assignment 3

March 29, 2021

Question 1.

Name: Braeden Hall
Student Number: 101143403

Question 2.

Algorithm 1 FindCycle:

```
function CYCLE($e = \{x, y\}, A$)
    for $i = 0$ to $|V|$ do
        $visited(v_i) \leftarrow false$
    end for
    $visited(y) \leftarrow true$

    for each edge $\{y, u\} \in A[y]$ do
        if $u = x$ then
            continue
        end if
        if $visited(u) = false$ then
            if $explore'(u, x, A) = true$ then
                return true
            end if
        end if
    end for

    return false
end function

function $explore'(u, x, A)$
    $visited(u) \leftarrow true$
    for each edge $\{u, v\} \in A[u]$ do
        if $v = x$ then
            return true
        end if
        if $visited(v) = false$ then
            if $explore'(v, x, A) = true$ then
                return true
            end if
        end if
    end for

    return false
end function
```

Proof of Correctness:
The algorithm starts the same as the DFS algorithm we have seen in class, by setting the visited value of each vertex $v \in V$ to false. It then sets the visited value for the second vertex of the given edge $e = \{x, y\}$ to true. Then for every edge $\{y, u\}$ except e, it calls a modified version of the explore function we have seen in class on the vertex u.

This modified explore function takes the vertex to be explored (u) and the other vertex of the edge e (x) as arguments. After setting the visited value to true for the current vertex u, the function looks at every edge with u as a vertex and recursively calls explore' on the other vertex (assuming it has not already been explored). If the other vertex of this edge happens to be x then we have found a cycle with the edge e and we return true cascading up until we get back to the cycle function and return true.

We know the cycle has been found because the original u (u_1) has an edge connecting it to y. There is then some sequence of edges $\{u_1, u_2\}$, $\{u_2, u_3\}, \ldots,$ $\{u_k, x\}$. Since $\{x, y\}$ is the edge we are looking for, as soon as we see the vertex x as part of some edge that we have found a cycle (explore' y will never be called since we set the visited value at the beginning of the algorithm).

If no call to explore' ever looks at an edge with x as a vertex then the second for loop in the cycle function terminates and the function returns false as it should since no cycle was found.

Proof of Runtime:
In the worst case, the algorithm will call explore' on every vertex exactly once. Since the runtime of the for loop in that call to explore' takes time proportional to the degree of that vertex, this will lead to each edge being looked at exactly twice. So the time the algorithm will take to finish is $\leq |V| + 2|E|$. Therefore, the algorithm runs in $O(|V| + |E|)$ time.

Question 3.

3.1: Please view Figure 1 in the appendix

3.2: Please view Figure 2 in the appendix

Question 4.

1. The algorithm ShortestPathAcyclic correctly computes the longest path from the source vertex to every vertex v for an arbitrary DAG.

   Let $\delta(s, v_i)$ denote the length of the longest path from s to vertex v_i for each $v_i \in V$.

   Claim 1: At any moment, $-\delta(s, v_i) \leq d(v_i)$.
   Proof: Given by definition of longest path.

   Claim 2: Assume that if some moment $d(v_i) = -\delta(s, v_i)$. Then, afterwards $d(v_i)$ does not change.
   Proof: In the algorithm given, if $d(v_i)$ changes it gets smaller. By claim 1, $d(v_i)$ can never be less than $-\delta(s, v_i)$.

   Claim 3: Take the longest path from s to v_i. This path has a last edge (v_j, v_i) where $j < i$. If at the start of iteration j $d(v_j) = -\delta(s, v_j)$ then at end of iteration j $d(v_i) = -\delta(s, v_i)$.
   Proof:

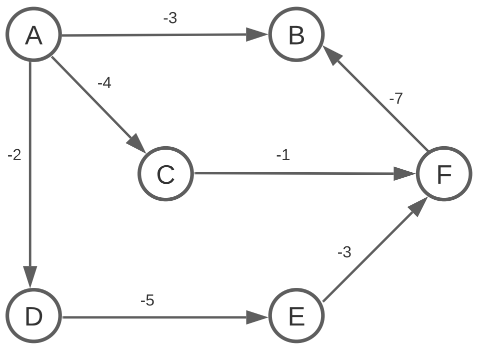
$$\begin{aligned} d(v_i) &\leq d(v_j) + wt'(v_j, v_i) \\ &= -\delta(s, v_j) + wt'(v_j, v_i) \\ &= -\delta(s, v_i) \leq d(v_i) \qquad \text{By claim 1} \end{aligned}$$

   Therefore $d(v_i) = -\delta(s, v_i)$.

Claim 4: At the end of the algorithm, for every $1 \leq i \leq |V|$: $d(v_i) = -\delta(s, v_i)$.
Proof: This claim is obviously true for $i = 1$. Since the algorithm progresses incrementally from v_1 to v_2 to v_3 etc., by claim 3 this claim is true for every $2 \leq i \leq |V|$. By claim 2, once some $d(v_i)$ becomes equal to $-\delta(s, v_i)$ it never changes. Therefore, this claim must be true.

2. The Dijkstra's algorithm doe not correctly compute the longest path from the source vertex to every vertex v for an arbitrary DAG.

   The following is a counter example:



   Assuming A is the source vertex, running Dijkstra on this graph to find the longest path would result in the following values for the longest paths (after negating the negative values):
   $d(A) = 0$; Path: A,
   $d(B) = 12$; Path: A->C->F->B,
   $d(C) = 4$; Path: A->C,
   $d(D) = 2$; Path: A->D,
   $d(E) = 7$; Path: A->D->E,
   $d(F) = 10$; Path: A->D->E->F

   All of these values are correct except the value for B. The longest path to B is: A->D->E->F->B and $\delta(A, B) = 17$. The algorithm gets this wrong because the value $d(B)$ is only updated when F is chosen from the set Q, which happens when $d(F) = -5$. The value of $d(F)$ is later updated to -10. However, since F is no longer in Q at this point the value $d(B)$ will never be updated to the correct value of -17.

Question 5.

If we look at Kruskal's approach to finding the MST, we can see that after sorting the edges by weight the algorithm first divides each vertex into a separate set/list. It then looks through the sorted edges in order and finds the sets/lists that each vertex of the edge is contained in. If they are both in the same list it does nothing, if they are in separate lists it joins the lists together. Obviously the edge with the smallest weight must be in the MST since the all vertices are in separate lists at the start. Since the graph is simple, I argue that the edge with the second smallest weight must also be in the MST.

(5.1) Assume that the first edge connected vertices A and B together. It is not possible for another edge to directly connect both of these edges together; i.e. the second edge must either connect one of these vertices to a separate vertex, or connect two different vertices together. Either way the edge with the second smallest weight must be part of the MST. When it comes to the third edge it is a different story.

(5.2) Following from the previous example lets assume that the second edge connected vertex B to vertex C. Now we have a tree that contains the vertices A, B and C. If we also assume that the third smallest edge is the edge {A,C}, then when the algorithm checks it will find that the vertices A and C are already part of the same list. This edge will effectively be skipped and not added to the MST. Therefore, it is not necessarily true that the edge with the third smallest weight is part of the MST.

Question 6.

Data structure is as described in the question.
Algorithm for $Find(x)$ does not change since the list data structure still gives each element access to the head of the list which stores the lists name.
Algorithm for new $Union(A, B, C)$ is as follows:

Algorithm 2 $Union(A, B, C)$:

```
$x \leftarrow A.head.next$
$y \leftarrow B.head.next$

while $x.next \neq NULL$ and $y.next \neq NULL$ do
    $x \leftarrow x.next$
    $y \leftarrow y.next$
end while

if $x = A.last$ then                                        ▷ *
    $C \leftarrow B$
    $C.last.next \leftarrow A.head.next$
    $C.last \leftarrow A.last$
    for each element $e$ of $A$ do
        $e.back \leftarrow C.head$
    end for
else                                                        ▷ **
    $C \leftarrow A$
    $C.last.next \leftarrow B.head.next$
    $C.last \leftarrow B.last$
    for each element $e$ of $B$ do
        $e.back \leftarrow C.head$
    end for
end if
```

* If the while loop terminated because $x.next$ was $NULL$ then x will be equal to $A.last$
** This is equivalent to if $y = B.last$ because that is the only other way the while loop terminates

Proof of Correctness:
The first thing the algorithm does is decide which list is smaller; this is the same as what is done in the original Union algorithm just done in a different way since the size of each list is no longer stored in its head. The second step is to append the shorter list to the end of the long one and set the result to C. This is also the same as what happens in the original Union algorithm seen in class.

Proof or Runtime:
The algorithm starts by setting the elements x and y to the first elements in the lists A and B respectively. It then loops through both lists incrementally until one of them (the shorter one) has no more elements to iterate over. As this the loop only iterates over one of the lists to completion it therefore takes $O(min(|A|, |B|))$ time. The algorithm then sets C equal to the the longer list, sets the new last pointer to the end of the shorter list and finally iterates over the shorter list's elements and sets their back pointers to the the new list head. This effectively merges the 2 lists A and B. Since the for loop will only iterate over the shorter list,

this too will take $O(min(|A|, |B|))$ time. Therefore, as the highest time complexity of any operation done by the new Union algorithm is $min(|A|, |B|)$, the algorithm runs in $O(min(|A|, |B|))$.

Question 7.

Recurrence:

```
for $0 \leq i \leq m$, $0 \leq j \leq n$ do
    Let $L[i, j]$ = the length of the longest common sub-string of $A$ and $B$ that ends at $a_i$ and $b_j$
    if $i = 0$ or $j = 0$ then
        $L[i, j] = 0$
    end if
    if $a_i = b_j$ then
        $L[i, j] = L[i - 1, j - 1] + 1$
    else
        $L[i, j] = 0$
    end if
end for
```

Proof of Recurrence:
The first if statement is trivial; there is no bit for a_0 or b_0 so there can be no sub-string that ends at either of these locations and $L[i, j]$ must be 0 at these points. The second if statement follows from the fact that if a_i and b_j are equal then the length of the common sub-string that ends at a_i and b_j is 1 longer than the length of the common sub-string that ends at a_{i-1} and b_{j-1}. The else condition follows from the hint; if the bits themselves are not common (equal) then their can be no common string ending at those bits.

We can think of L as a matrix with j columns and i rows. Since the value of $L[i, j]$ depends on the that of smaller values of i and j it makes sense to calculate these values iteratively left to right and working down the matrix.

Algorithm 3 $LCS'(A, B)$:

```
 1: $m \leftarrow |A|$, $n \leftarrow |B|$
 2: $x \leftarrow 0$, $y \leftarrow 0$
 3: $length \leftarrow 0$
 4: for $i = 0$ to $m$ do
 5:     $L[i, 0] \leftarrow 0$
 6: end for
 7: for $j = 1$ to $n$ do
 8:     $L[0, j] \leftarrow 0$
 9: end for

10: for $i = 1$ to $m$ do
11:     for $j = 1$ to $n$ do
12:         if $a_i = b_j$ then
13:             $L[i, j] \leftarrow L[i-1, j-1] + 1$

14:             if $L[i, j] > length$ then
15:                 $x \leftarrow i$, $y \leftarrow j$
16:                 $length \leftarrow L[i, j]$
17:             end if
18:         else
19:             $L[i, j] \leftarrow 0$
20:         end if
21:     end for
22: end for

23: $string = ""$
24: for $i = x - length + 1$ to $x$ do
25:     $string \leftarrow string + a_i$
26: end for

27: return $(length, string)$
```

Proof of Runtime:
The first 2 for loops initialize the first row and column of the matrix; this takes $O(m)$ and $O(n)$ time respectively. The for loop on line 10 runs for m iterations and each iteration the inner for loop iterates for n iterations. The statements inside these for loops all take constant time, so the runtime for the for loop on line 10 is $O(mn)$. The final for loop on line 24 iterates for *length* iterations to build the common sub-string. Therefore, $length \leq min(m, n)$; this means that the for loop takes $O(min(m, n))$ time. Since the highest time complexity of any operation in the algorithm is $O(mn)$, therefore, the algorithm runs in $O(mn)$ time.

Appendix

Figure 1:

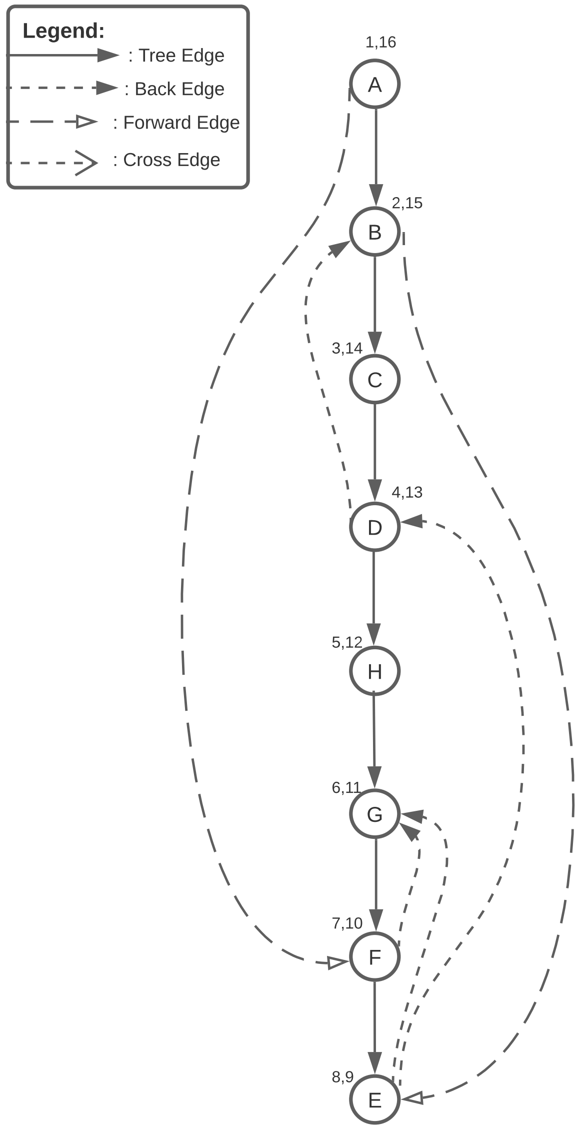
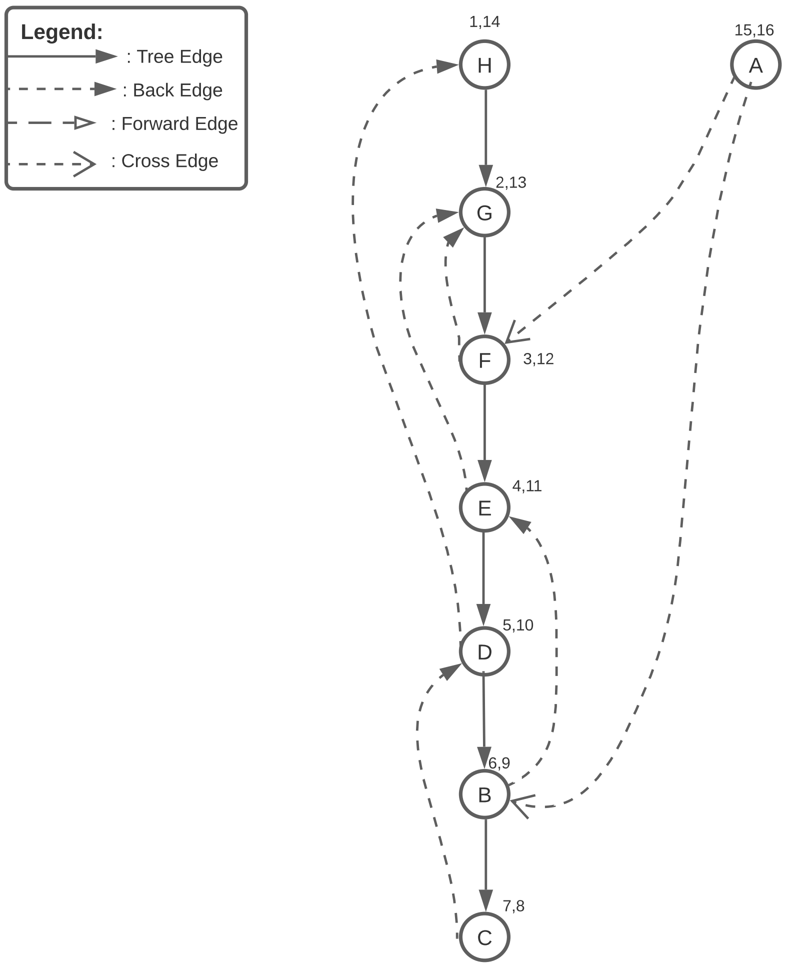


Figure 2:


Legend:
: Tree Edge
: Back Edge
: Forward Edge
: Cross Edge
1,14
H
15,16
A
2,13
G
F
3,12
4,11
E
5,10
D
6,9
B
7,8
C