Atanas Dimtirov
Homework #4
CS 6470
Dr. Cai

Solution to problem 1:
Input: Directed or undirected graph G

Output: Same as DFS search but in addition all edges' types are printed as tree, forward, cross, or back

Algorithm:

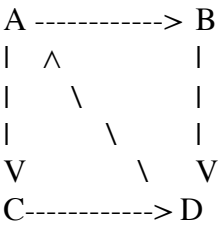
```
Depth_First_Search_Print_Edges(G)
        for each vertex u ∈ V[G]
                do color[u] ← WHITE
                        ∏[u] ← NIL
        time ← 0
        for each vertex u ∈ V[G]
                do if color[u] ← WHITE
                        then DFS-VISIT(u)

DFS-VISIT(u)
        color[u] ← GRAY
        time=time+1
        d[u]← time
        for each v ∈ Adj[u]
                do if color[v] ← WHITE
                        then    ∏[v] ← u
                                print that u,v is a tree edge
                                DFS-VISIT(v)
                if color[v] ← BLACK
                        then    print that u,v is a forward edge
                        else    print that u,v is a cross edge
                if color[v] ← GRAY
                        then    print that u,v is a back edge
        color[u] ← BLACK
        f[u] ← time ← time +1
```

As specified in the textbook *p. 546* :
1. *White* - tree edge
2. *Gray* - back edge
3. *Black* - forward or cross

Thus in order to accomplish the printing of every edge we preserve the semantics of DFS and

simply add some checks associated with the color of the node. Since in an undirected graph is either a tree or a back edge as proven in *Theorem 22.10* we are assured that the if statement involving the BLACK color check is not going to be executed. Thus no modifications are necessary but the lines containing the associated pseudo code can be taken out.

Solution to problem 2:
The answer is no. The topological sort doesn't necessarily print out ordering that minimizes the number of "bad" edges for a graph G which contains cycles every time. We can simply provide a counter example. The general idea is that the algorithm will print out a different ordering every time we choose a different node to be considered as the entry point.
Suppose we have the graph:



Or:

A: B, C
B: D
C: D
D: A

Thus the graph contains 2 cycles: ACD and ABD
Starting from A we get the following ordering: ABCD 1 "bad" edge
But starting from D we get: DABC 2 "bad" edges
Thus we have provided a counter example which disproves the statement making it FALSE.

Solution to problem 3:
Since we want to only select vertices with in-degree 0 we must create a data structure to keep track of this. We can also use the data structure to select which vertex is to be processed next, in other words the data structure must be able to point out which is the next vertex with in-degree 0.
We can use the following algorithm to accomplish the topological sort:

1. We calculate the in-degree of each vertex as follows:

- use array in-degree-value[i] to store the in-degree value of vertex i
- if vertex i is in the adj. list of vertex j then increase the value of in-degree-value[i] by 1.
- do this until there are no more vertices in the adj. list of j then move to the next vertex and process its adj. list.

Since we are considering at most |V| vertices and at most |E| edges this will take O(E+V)

2. Initialize the data structure and process the vertices with in-degree=0.

- go through each entry in in-degree-value[] and enqueue (in a generic queue Q) each vertex with in-degree=0.

This will take O(V) since each vertex will be considered once and the enqueue operation is O(1).

- dequeue 1 element(vertex) u from the queue and for each vertex v in its adj. list decrease its in-degree-value[v] by 1 for each time it v appears in the adj list of u.
- if after decreasing in-degree-value[v], in-degree-value[v] =0 then enqueue vertex v.
- print out vertex u

De-queueing will take O(1) - we take the first vertex on the queue. Each edge will be considered at most once and it takes O(1) to remove the edge (simply reduce the value of in-degree-value[v] by 1)- then the total number of edge "removals" is O(E). Thus the total number of operations associated with step 2 all-together is O(E+V).

Since each vertex v from an edge (u,v) can only be output whenever u has been already considered, thus it has been output, then we can clearly see that this algorithm produces the output correctly ordered since this implies u output before v.
If the graph is acyclic then we can clearly see that all vertices will be output because if they are not this will imply a cycle. If there is a cycle then some vertices will never have their in-degree-value=0.

Solution to problem 4:

When all the edge weights are integers from 1 to |V|:
This setting can make a difference in terms of the time required to sort the edges. Initially, as stated in the textbook *p. 570*, we have total of O(ElogE) search time. It seems that this term eventually dominates the overall time complexity of the algorithm. To be more precise the time required according to the textbook is O(1) + O(ElogE) + O(E*alpha(V)). As seen in *Section 8.2, p. 169* Counting sort has time complexity of O(k+n) where k is the integer range of the values in the input list and n is the nnumber of input elements. We can apply this algorithm to sort the edges which will give us: O(V+E). Since G is assumed connected then $|E| \geq |V|-1$ thus V=O(E). It follows that sorting can be done in O(V+E)=O(E+E)=O(E). The overall time complexity then becomes O(E*alpha(V)) be replacing O(ElogE) by O(E).

When all edge weights are integers between 1 and constant W we can use the same method this time sorting will take O(W+E) which will eventually become O(E) thus the overall time complexity will be still O(E*alpha(V)) be replacing O(ElogE) by O(E).

Solution to problem 5:

When all the edge weights are integers from 1 to W:
It is possible to change the data structure for storing the elements in order to reduce the time for some of the operations associated initially with the min-heap. We can use a technique which somewhat resembles bucket sort. If we say have W buckets to hold edges of length 1 through W respectively we can easily go through them and extract the minimal weight edge since we only have to start from bucket 1 and check if we have any elements in it. Each bucket can be implemented as a linked list of elements with the same weight. The set of buckets can be implemented as an array. Each of the arrays are a pointer to the respective linked list. In this case we will have O(W) time for finding the next shortest edge or the equivalent for EXTRACT-MIN. Since W is constant then O(W)=O(1). The time complexity of DECREASE-KEY can also be reduced with this setup since in this case it will only take O(1) - moving an element from 1 bucket to another. Thus the overall

time complexity initially shown in the book *p. 573* as O(V) + O(VlgV) + O(ElgV) will change to O(V) + O(V) + O(E)=O(V)+O(E)= O(V+E). If we assume G is connected then |E|≧|V|-1 thus the time complexity becomes O(E).

When all edges range from 1 to |V|:
As described on *p. 144* van Emde Boas priority queue will take O(lglgV) to perform EXTRACT-MIN and DECREASE-KEY operations. The overall time complexity thus becomes O(V) + O(V*lglgV) + O(ElglgV). Assuming the graph is connected - |E|≧||V|-1 - We have O(VlglgV + ElglgV) and subsequently this equals O(ElglgV).