CMSC351 - Fall 2014, Homework #5

Due: December first at the start of class

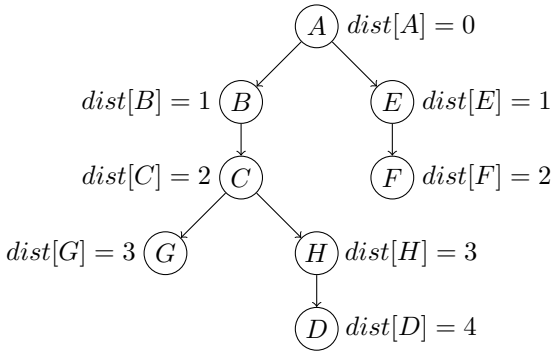
PRINT Name:

- Grades depend on neatness and clarity.
- Write your answers with enough detail about your approach and concepts used, so that the grader will be able to understand it easily. You should ALWAYS prove the correctness of your algorithms either directly or by referring to a proof in the book.
- Write your answers in the spaces provided. If needed, attach other pages.
- The grades would be out of 100. Four problems would be selected and everyones grade would be based only on those problems. You will also get 25 bonus points for trying to solve all problems.

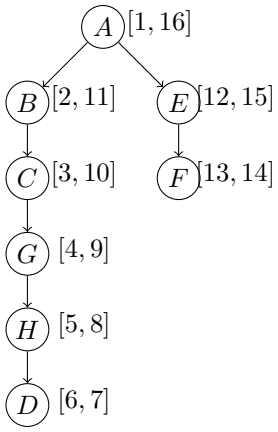
Note: In this problem set, n denotes the number of vertices and m denotes the number of edges. By a linear algorithm, we mean an algorithm that runs in $O(m+n)$.

Problem 1 (a) Run BFS algorithm on the directed graph below, using vertex A as the source. Show all distances and the BFS tree.

(b) Run DFS algorithm on the directed graph below, using vertex A as the source. Compute the start time and the finish time of each vertex and draw the DFS tree.



(a) BFS tree.



(b) DFS tree

Problem 2 **(CLRS 22.1-6)** Most graph algorithms that take an adjacency-matrix representation as input require time $O(n^2)$, but there are some exceptions. Show how to determine whether a directed graph G contains a *universal sink*, i.e., a vertex with in-degree $n-1$ and out-degree 0, in time $O(n)$ given an adjacency matrix for G.

Solution Suppose you have matrix A, the adjacency matrix of the graph, where $A(i,j)$ is 1 if and only if i has a directed edge to j. If a vertex v is a *universal sink* in the graph, all the other vertices have an edge to it and it has no edges to other vertices. This means the row corresponding to vertex v is all 0 in matrix A, and the column corresponding to vertex v in matrix A is all 1 except for $A(v,v)$. Needless to say, there is at most one universal sink in the graph.

Now suppose we are looking at cell $A(i,j)$ for $i \neq j$. If $A(i,j) = 0$, it means vertex i does not have an edge to vertex j. This means vertex j cannot be a universal sink. On the other hand, if $A(i,j) = 1$, it means vertex i has an edge to vertex j, and thus vertex i cannot be a universal sink. Therefore, by looking at each cell of the matrix, you can remove one vertex from the set of potential universal sinks.

The algorithm for finding the universal sink of the graph is as follows. Start with all vertices in the set of potential universal sinks S. Each time pick two different vertices x and y from this set and look at the cell $A(x,y)$. If $A(x,y)$ is 1, remove x from S, and otherwise remove y from S. Therefore, in $n-1$ steps, and by looking at one cell of the matrix at each step, you can removed $n-1$ vertices from S and you only have one potential universal sink say vertex u. Check the whole column and row corresponding to vertex u in matrix A and declare u as a universal sink if the whole row is zero and the whole column is one. Otherwise, the graph does not have a universal sink. In this Algorithm, we check $n-1$ cells of the matrix to end up with one vertex and then we check the whole row and column corresponding to that vertex for a total of $(n-1)+(2n-1)$ cells. Hence, the algorithm has $O(n)$ running time.

Problem 3 Suppose we have an undirected graph and we want to color all the vertices with two colors *red* and *blue* such that no two neighbors have the same color. Design an $O(n+m)$ time algorithm which finds such a coloring if possible or determines that there is no such coloring.

(a) Prove that if the graph has a cycle of odd length, there is no such a coloring.

(b) Solution : Suppose the graph has an odd cycle but a valid coloring. Let $v_1, v_2, \ldots, v_k$ be the vertices of the cycle. Without loss of generality suppose v_1 is colored blue. Then v_2 must be red, v_3 must be blue and so on. Therefore, any v_i should be colored blue if i is odd and should be colored red if i is even. Since k is odd, v_k must be colored blue. However, v_k and v_1 are neighbors and both are blue which is a contradiction.

(c) Assume the graph has no cycle of odd length. Use BFS algorithm to find an appropriate coloring.

Solution : First of all, note that if there is a valid coloring where vertex 1 is colored red, then you can swap all the colors of vertices to get a valid coloring of the vertices in which vertex 1 is blue. Therefore, we can assume that if there is a valid coloring of the vertices, there is also a valid coloring in which vertex 1 is blue. Furthermore, we can assume that the graph is connected (color each connected component separately).

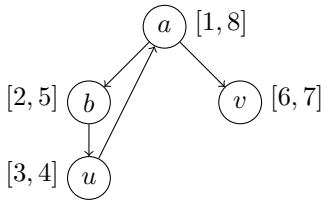
Suppose vertex 1 is blue. Run BFS from vertex 1. If there is a valid coloring, all the vertices in level one of BFS *must be* colored red (since they are neighbors of vertex 1 which is blue). Similarly, all the vertices of level 2 *must be* blue. All the vertices of level 3 must be red and so on. You can color all the vertices of the graph based on the parity of their distance from vertex 1. The order of algorithm is the same as BFS; i.e, $O(n+m)$.

Now we want to show that if there is no odd cycle, then this algorithm gives an appropriate coloring. Consider the BFS tree rooted at vertex 1. If our algorithm does not produce a valid coloring, then there is an edge in the graph with endpoints u and v such that u and v have the same color. Let $d(u)$ and $d(v)$ denote the distance of vertices u and v from vertex 1. If u and v are colored the same, then $d(u)$ and $d(v)$ must have the same parity (either both are odd or both are even). Now consider the shortest path from vertex 1 to vertex u and the shortest path from vertex 1 to vertex v along with the edge between u and v. They form a cycle of length $d(u) + d(v) + 1$ which is odd (since $d(u)$ and $d(v)$ are same parity). This is a contradiction to the fact that the graph does not have an odd cycle. Therefore, our algorithm is producing a valid coloring.

Problem 4 (a) **(CLRS 22.3-9)** Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.start \leq u.finish$.

(b) **(CLRS 22.3-8)** Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, and if $u.start < v.start$ in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.

Solution: Consider the following graph as a counterexample for both parts. Note that in this example there is a path from u to v. Run DFS from vertex a. Consider b as the first neighbor of a during DFS, and u as a neighbor of b. So, we would have $u.start = 3$, $u.finish = 4$, $v.start = 6$, and $v.finish = 7$.



Problem 5 **(CLRS 22.3-7)** Implement DFS using a stack to eliminate recursion. Write a pseudo-code for your algorithm.

Solution : Let L be the adjacency list of the graph, and v be the vertex that you want to start dfs from. We want to simulate the recursive function using a stack. Whenever we are considering a vertex x in dfs, we must move along all neighbors of x and see if they have not been visited before. As soon as we find one vertex satisfying this condition, we stop and run dfs from that vertex. We keep track of the vertices that we have considered for vertex x so far by a variable $head[x]$. At each time, if we are running dfs from vertex x, we try to update $head[x]$ to get to a vertex that is adjacent to x and not visited before. If we find such a vertex, we push it to the stack so that it would be the next vertex that we consider in the next step. Once $head[x]$ is equal to $null$, it means that we have considered all the vertices for vertex x and dfs for this vertex is finished. At this time we can pop x from the stack.

Algorithm 1 DFS

```
1:  S.push(v)                                   ▷ v is the vertex we begin dfs from
2:  color[v] = gray
3:  Let head[x] be the first neighbor of x in its corresponding linked list.
4:  while S is not empty do
5:      x ← S.top
6:      while head[x] ≠ null and color[head[x]] ≠ white do
7:          head[x] = head[x].next
8:      end while
9:    ▷ In case we have found the unvisited adjacent vertex head[x] do the following
10:     if head[x] ≠ null then          ▷ Then start dfs from head[x] in the next step
11:         S.push(head[x])
12:         visited[head[x]] = gray
13:     else                                ▷ If no such vertex, dfs is finished for x
14:         S.pop
15:         color[x] = black
16:     end if
17: end while
```

Problem 6 Given a directed acyclic graph G, design an $O(n+m)$ time algorithm which finds the length of the longest path of the graph.

(a) Find a topological sort of the given DAG and let $v_1, v_2, \ldots, v_n$ be a topological sort, i.e., each edge is from a vertex v_i to another vertex v_j with $j > i$. Let $A[i]$ be the longest path of the graph starting at v_i. Find a formula for computing $A[i]$.

Solution :

$$A[i] = \max_{j>i \ \& \ (i,j)\in E} A[j] + 1 \tag{1}$$

(b) If we compute $A[1], A[2], \ldots, A[n]$, what would be the final solution?

Solution : $\max_{i=1,\ldots,n} A[i]$

(c) Write a dynamic program for filling array A. What is the running time of this algorithm?

Solution :

```
1: for i ← n to 1 do
2:     A[i] = 0
3:     for all out-neighbors of v_i such as v_j do
4:         A[i] = max{A[i], A[j] + 1}
5:     end for
6: end for
```

For each vertex i, we check each of its edges exactly once. Therefore, the algorithm is $O(n+m)$.

(d) Run DFS and compute $A[i]$ at the end of DFS(v_i). How do you compute $A[i]$ at the end of DFS procedure? What is the running time of this algorithm?

Algorithm 2 DFS

```
1: procedure DFS(u)
2:     color[u] = gray
3:     A[u] ← 0
4:     for all nieghbors of u such as v do
5:         if color[v] = white then
6:             DFS(v)
7:         end if
8:         A[u] = max(A[u], A[v] + 1).
9:     end for
10: end procedure
```

The running time is equal to the running time of DFS which is $O(n+m)$.

Problem 7 (CLRS 22.4-5) Another way to perform topological sorting on a directed acyclic graph G is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(n + m)$.

Solution : First compute all vertices in-degrees by making a pass through all the edges and incrementing the in-degree of each vertex an edge enters. This takes $O(n + m)$ time.

Let Q be a first in first out queue. First push all the vertices with in-degree zero to Q (the order does not matter).

As we process each vertex from the queue, we effectively remove its outgoing edges from the graph by decrementing the in-degree of each vertex one of those edges enters, and we enqueue any vertex whose in-degree becomes 0. Each vertex is en-queued/dequeued at most once because it is enqueued only if it starts out with in-degree 0 or if its in-degree becomes 0 after being decremented some number of times.

Note that each vertex appears exactly once in Q. Because the adjacency list of each vertex is scanned only when the vertex is dequeued, the adjacency list of each vertex is scanned at most once. Since the sum of the lengths of all the adjacency lists is $O(m)$. Furthermore, each vertex is enqueued and dequeued exactly once which takes $O(n)$ time. Therefore, the algorithm runs in $O(n+m)$ time.

To get the Topological sort of the vertices, rank the vertices by their order of appearance in Q. Each vertex v appears in the queue once its in-degree becomes zero. This means all of its parents (vertices that have an edge to v) must have been in Q before. Therefore, if we consider the vertices in the order of their appearance in Q, there would be no backward edges. Hence this ordering is a valid Topological sort of the vertices of the graph.

Problem 8 (CLRS 22.2-8) The *diameter* of a tree $T = (V, E)$ is defined as the length of the longest path in the tree. Give an $O(n)$ algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

Hint 1: We have $m = n - 1$ in a tree, and thus $n + m = \Theta(n)$.

Hint 2: Pick an arbitrary vertex r as the root. Run DFS algorithm from r. For each vertex i define $A[i]$ as the length of the longest path in the subtree rooted as i, and $B[i]$ as the length of the longest path in the subtree rooted as i which has vertex i as one of its end points. Show how to compute $A[i]$ and $B[i]$ for each vertex i at the end of DFS(i).

Solution : Consider the graph(tree) rooted at vertex r. First let's see how to compute $B[i]$ for a vertex i given $B[j]$ for all $j \in children(i)$. It is the length of the longest path in the subtree rooted at i which has i as one of its endpoints. So the path must start at i and go through one of its children (or end at i in case i has no children). Since we want to maximize $B[i]$, it is better to go through the child which has longest path starting from it. That is we should pick child j which has maximum $B[j]$ among all children of i. Therefore, $B[i] = \max_{j \in children(i)} \{B[j] + 1\}$.

Now suppose we want to calculate $A[i]$ of vertex i assuming we have all $A[j]$ and $B[j]$ for any other vertex in subtree rooted at i. The longest path in the subtree rooted at i can either be completely in a subtree rooted at one of its children, or it can be a path going through vertex i. If it is completely in one of its children's subtrees we are interested in $\max_{j \in children(i)} \{A[j]\}$. Otherwise, i is in the path.

Consider the path going through vertex i in the subtree rooted at i. It is going through zero, one or two of vertex i's children depending on the number of children vertex i has. Let's say it goes through vertices $j, j' \in children(i)$. The length of the longest path passing through i, j, j' is equal to $B[j] + B[j'] + 2$. Let Max_1 and Max_2 be the maximum and second maximum of $B[j]$ among all $j \in children(i)$. If i has no children, then $Max_1 = Max_2 = -1$. If i has only one child j, then Max_1 is $B[j]$ and $Max_2 = -1$. Otherwise, Max_1 and Max_2 are the maximum and second maximum of $B[j]$'s respectively. The length of the longest path passing through vertex i would be equal to $Max_1 + Max_2 + 2$. Therefore, $A[i] = \max\{Max_1 + Max_2 + 2, \max_{j \in children(i)} \{A[j]\}\}$.

By definition, $A[r]$ is the length of the longest path in subtree rooted from r which is the whole graph. The length of the longest path in a graph is the diameter of the graph.

Algorithm 3 DFS

```
1: procedure DFS(x)
2:     visited(x) ← 1
3:     A[x] ← 0
4:     B[x] ← 0
5:     for y ∈ N(x) do
6:         if visited(y) = 0 then
7:             DFS(y)
8:             A[x] = max(A[x], A[y]).
9:             B[x] = max(B[x], B[y]).
10:            Update Max_1, Max_2 with B[y]
11:        end if
12:    end for
13:    A[x] = max(A[x], Max_1 + Max_2 + 2)
14: end procedure
```