Solution of Forth Homework

Han Wu, Yu Chen

1 Examples (5 points)

Give one example of a directed graph on four vertices, A, B, C, and D, so that both depth-first search and breadth-first search discover the vertices in the same order when started at A. Give one example of an directed graph where BFS and DFS discover the vertices in a different order when started at A. Above, discover means the time that the algorithm first reaches the vertex. Assume that both DFS and BFS iterate over outgoing neighbors in alphabetical order.

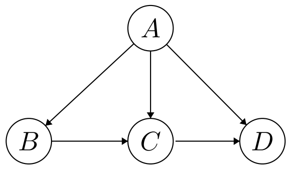

Figure 2: Same Order Example

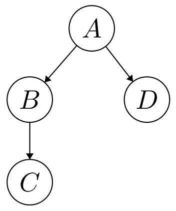

Figure 3: Different Order Example

Figure 2 shows an example of BFS traversing in the same order as DFS. Their traversal order is $\langle A, B, C, D\rangle$. Figure 3 shows an example of BFS traversing in a different orde with DFS. The traversal order of BFS and DFS are $\langle A, B, D, C\rangle$ and $\langle A, B, C, D\rangle$ respectively.

2 In-order traversal (10 points)

In class, we stated the fact that Depth-First Search can be used to do in-order traversal of a binary search tree (BST). That is, given a binary search tree T containing n distinct elements $a_1 < a_2 < \cdots < a_n$, DFS can be used to return an array $[a_1, a_2, \ldots, a_n]$ containing these elements in sorted order. Call this algorithm **`inOrderTraversal`**. Work out the details for this algorithm, and answer the following questions:

(a) (5 pts) Give pseudocode for **`inOrderTraversal`**. For your pseudocode, assume that each node in T has a key value (one of the a_i) and pointers to its left and right children, and that if a vertex has no child this is represented as a **`NIL`** child.
As shown in algorithm 1.

Algorithm 1 inOrderTraversal(*root*)

```
1: if $root == NIL$ then
2:     return
3: inOrderTraversal(root.left)
4: print(root.val)
5: inOrderTraversal(root.right)
```

(b) (3 pts) What is the asymptotic running time of **inOrderTraversal**, in terms of n? We're looking for a statement of the form "the running time is $\Theta(\dots)$." [Hints: We notice that each node in a Tree has only one parent.]
The running time is $\Theta(n)$.

(c) (2 pts) Does the algorithm need to look at the values $a_1, a_2, \dots, a_n$ (other than to return the values)? In other words, what will affect the procedure of your **inOrderTraversal**, the values of $a_1, a_2, \dots, a_n$ or the structure of the BST?
Affected by the structure of BST.

3 Level Averages for a Given Binary Tree (5 points)

For a binary tree T, design an algorithm **levelAverage** to report the average values for each level of it in a descending order of levels (i.e., reporting the average value for the deepest level first). Give the pseudocode for **levelAverage**.
As shown in algorithm 2.

Algorithm 2 levelAverage(*root*)

```
 1: averages ← initial a list for storing averages
 2: queue ← initial a queue for level traversal
 3: queue.push(root)
 4: while !queue.empty() do
 5:     len ← queue.size()
 6:     for i in 0..len do
 7:         node ← queue.poll();
 8:         sum+ = node.val
 9:         if node.left! = NIL then
10:             queue.push(node.left)
11:         if node.right! = NIL then
12:             queue.push(node.right)
        averages.add(sum/len)
13: return averages
```

4 Using DFS to topologically sort nodes in a directed graph (5 points)

A formal definition for the topological ordering of a graph is: for a directed graph G, we say that a topological ordering of G is an ordering of its nodes as v_1, v_2, . . . , v_n so that for every edge (v_i, v_j) (i.e., a direct edge pointing from v_i to v_j), we have $i < j$. Now we have a directed graph shown in Figure 3, we can use the DFS-based algorithm introduced in Lecture 9 to topologically sort the nodes in it. However, as the start node is randomly picked, the reported topological ordering may be different. For the following orders, which one(s) is/are correct? For each correct topological order, please report an ordered list of vertices discovered by our algorithm.

(a) a, b, c, d, e

(b) a, b, c, e, d

(c) a, c, b, d, e

(d) a, d, b, c, e

(e) a, c, d, b, e

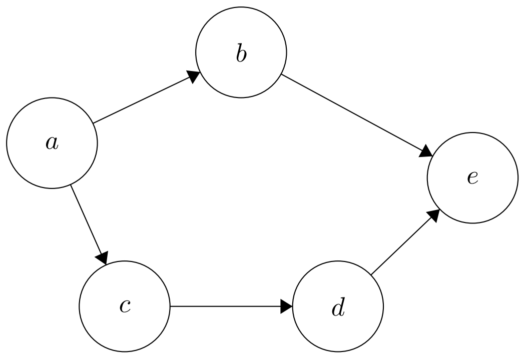

Figure 3: A directed graph.

By the definition of topolocial ordering, (a),(c),(d) are correct. (b) is incorrect because e is in front of d, and (d) is incorrect because d is in front of c.
One of possible orders discovered by our algorithm is: (a)(a,c,d,e,b), (b)(a,b,e,c,d), (c)(d,e,a,b,c). The start and finish time labeled by our algorithm are shown as follows:

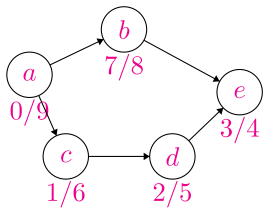

Figure 5: traversal order of (a)

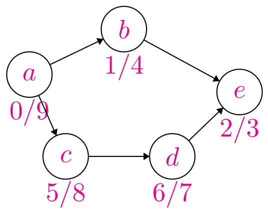

Figure 6: traversal order of (c)

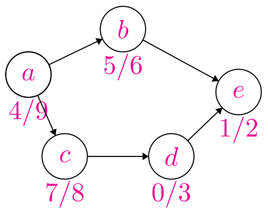

Figure 7: traversal order of (e)

5 Bipartite Graph Recognization (5 points)

Is the graph in Figure **??** a bipartite graph? Give your reason. [**We are expecting a brief explanation. No detailed proof is needed.**]
We can use coloring method to judge whether a graph is bipartite. And we can also use a property of bipartite graph to quickly judge whether a graph is bipartite: a bipartite graph will not have a ring with odd-length. In this figure, we can easily find that there is a ring $3-1-2-7-9-12-13-3$ whose length is odd. Thus, this graph is not bipartite.

6 Social engineering (10 points)

Suppose we have a community of n people. We can create a directed graph from this community as follows: the vertices are people, and there is a directed edge from person A to person B if A would forward a rumor to B. Assume that if there is an edge from A to B, then A will always forward any rumor they hear to B. Notice that this relationship isn't symmetric: A might gossip to B but not vice versa. Suppose there are m directed edges total, so $G = (V, E)$ is a graph with n vertices and m edges.

Define a person P to be *influential* if for all other people A in the community, there is a directed path from P to A in G. Thus, if you tell a rumor to an influential person P, eventually the rumor will reach everybody. You have a rumor that you'd like to spread, but you don't have time to tell more than one person, so you'd like to find an influential person to tell the rumor to.

In the following questions, assume that G is the directed graph representing the community, and that you have access to G as an array of adjacency lists: for each vertex v, in $O(1)$ time you can get a pointer to the head of the linked lists v`.outgoing_neighbors` and v`.incoming_neighbors`. Notice that G is not necessarily acyclic. In your answers, you may use to any statements or data structures we have seen in class, in the notes, or in CLRS (e.g., Red-Black Tree, Hash Table, BST and so on).

(a) (2 pts) Show that all influential people in G are in the same strongly connected component, and that everyone in this strongly connected component is influential.

[We are expecting: a short but formal proof.]

(b) (5 pts) Suppose that an influential person exists. Give an algorithm that, given G, finds an influential person in time $O(n + m)$.

[We are expecting: pseudocode, a proof of correctness, and a short argument about the runtime.]

(c) (3 pts) Suppose that you don't know whether or not an influential person exists. Use your algorithm from part (b) to give an algorithm that, given G, either finds an influential person in time $O(n + m)$ if there is one, or else returns "no influential person."

[We are expecting: pseudocode, and a short argument for both correctness and runtime. You do not need to re-write your algorithm from part (b), you can just call it.]

(a) We need to prove two things: the first one is all influential people are **in the same** strongly connected component, and the second one is everyone in this strongly connected component is influential. We will prove them by contradiction. For the first one, suppose there are two influential people P_a and P_b, P_a is in SCC_a and P_b is in SCC_b. Because they are in different SCC, thus there does not exists a bi-directional path between them, contradict to the fact that they are both influential people. Thus, the first one has been proved. For the second one, suppose there is a person P_c in the SCC and he/she is not an influential people, so there are some people in G cannot be reached by him/her. However, the other ones in the same SCC are all influential people, and there are bi-directional paths between P_c and the other ones in the SCC, so P_c can reach every other people in G, contradicted to our hypothesis. Thus, the second one has been proved.

(b) According to question (a), we can first find all SCCs in G(e.g.Tarjan), then check each SCC to find an influential people. Details are shown in algorithm 3;

(c) According to question (b), if the SCCs found by tarjan algorithm are not connected or there are more than one SCC with zero inDegree, then there isn't influential person exists.

Algorithm 3 Social engineering

```
1: $low \leftarrow$ an array of length $n$ ▷ $low[i]$ records the lowest dfs order of $i$'s neighbor, and the neighbor hasn't been added into another SCC(on the stack)
2: $ids \leftarrow$ an array of length $n$    ▷ $ids[i]$ records the dfs order of vertex $i$
3: $onStack \leftarrow$ an array of length $n$    ▷ $onStack[i]$ records whether $i$ is on stack or not
4: $stack \leftarrow$ a stack to store vertices
5: $id \leftarrow 0$    ▷ a global counter
6: $sccNum \leftarrow 0$    ▷ record the number of SCC
7: $scc \leftarrow$ a list    ▷ $scc[i]$ records a list of vertices that form a SCC
8: $id2scc \leftarrow$ a map    ▷ $id2scc[i]$ records the scc number of vertex $i$
9: procedure SocialEng($G, m, n$)
10:     for $i$ in $[0 : n)$ do
11:         if $ids[i] \neq 0$ then
12:             $dfs(i)$;
13:     $inDegrees \leftarrow$ an array of length $sccNum + 1$ ▷ $inDegree[i]$ records the inDegree of $i$th SCC. SCCs we find by tarjan algorithm will form a DAG(There must be an influential people, so all SCCs must be connected), and the SCC which contains influential people will have zero inDegree, otherwise, it can be merged with another SCC which has an edge pointing to it
14:     for $src$ in $G$ do
15:         for $dest$ in $G[src]$ do
16:             if $id2scc[src] \neq id2scc[dest]$ then
17:                 $inDegree[id2scc[dest]] + +$
18:     for $i$ in $[1 : sccNum]$ do
19:         if $inDegree[i] == 0$ then
20:             output $scc[i]$
21: procedure dfs($u$)    ▷ Tarjan algorithm to find all SCCs in $G$
22:     $stack.push(u)$
23:     $onStack[u] = true$
24:     $low[u] = ids[u] = + + id$
25:     for $v$ in $G[u]$ do    ▷ iterate through $u$'s neighbors
26:         if $ids[v] \neq 0$ then    ▷ $v$ hasn't been explored
27:             $dfs(v)$
28:         if ( then$onStack[v]$)    ▷ $u$'s neighbor $v$ is on stack, then update $u$'s low value
29:             $low[u] = min(low[u], low[v])$
30:     if $ids[u] == low[u]$ then    ▷ we find the start point of a SCC
31:         $sccNum + +$
32:         while $stack \neq \emptyset$ do
33:             $v = stack.pop()$
34:             $onStack[v] = false$
35:             $scc[sccNum].add(v)$
36:             $low[v] = ids[u]$
37:             $id2scc[v] = sccNum$
38:             if $v == u$ then
39:                 break
```