Lecture 21 Exercise Solutions

Mark Eramian

Exercise 1

Here's a trace through the algorithm.

Action	Queue (head to left)	Visited Nodes
Put node 0 on the queue to start.	0	-
Dequeue and visit 0.	-	0
Add 1, then 4 to the queue.	1, 4	0
Dequeue and visit 1.	4	0, 1
Add 2 to the queue.	4, 2	0, 1
Dequeue and visit 4.	2	0, 1, 4
Add 3, then 7 to the queue.	2, 3, 7	0, 1, 4
Dequeue and visit 2.	3, 7	0, 1, 4, 2
No unvisited nodes adjacent to 2.	3, 7	0, 1, 4, 2
Dequeue and visit 3.	7	0, 1, 4, 2, 3
Add 5 to queue.	7, 5	0, 1, 4, 2, 3
Dequeue and visit 7.	5	0, 1, 4, 2, 3, 7
No unvisited nodes adjacent to 7	5	0, 1, 4, 2, 3, 7
Dequeue and visit 5.	-	0, 1, 4, 2, 3, 7, 5
Add 6 to queue.	6	0, 1, 4, 2, 3, 7, 5
Dequeue and visit 6.	-	0, 1, 4, 2, 3, 7, 5, 6
No unvisited nodes adjacent to 6.	-	0, 1, 4, 2, 3, 7, 5, 6
Queue is empty, so stop.		

Thus, under the given assumptions, the nodes are visited in the order 0, 1, 4, 2, 3, 7, 5, 6.

Exercise 2

Here's a trace through the recursive calls of the algorithm.

```
call dftHelper(0)
    visit 0
    call dftHelper(1)
        visit 1
        call dftHelper(2)
            visit 2
            no unvisisted nodes adjacent to 2; return.
        no more unvisited nodes adjacent to 1; return.
    call dftHelper(4)
        visit 4
        call dftHelper(3)
            visit 3
            call dftHelper(5)
                visit 5
                call dftHelper(6)
                    visit 6
                    no unvisited nodes adjacent to 6; return.
                no more unvisited nodes adjacent to 5; return.
            no more unvisited nodes adjacent to 3; return.
        call dftHelper(7)
            visit 7
            no unvisited nodes adjacent to 7; return.
        no more unvisited nodes adjacent to 4; return.
    no more unvisisted nodes adjacent to 0; return.
```

Thus, under the given assumptions, the nodes are visited in the order 0, 1, 2, 4, 3, 5, 6, 7.

Exercise 3

```
// An algorithm for breadth first
// traversal of a graph
Algoirthm bft(s)
s is the start node in the graph

q = new Queue()

For each vertex v of V
    reached(v) = false

reached(s) = true
q.insert(s)

while not q.isEmpty() do
    w = q.item()     // get top node on stack
    q.deleteItem()   // pop the stack

    // perform the "visit" operation on w

    For each v adjacent to w do
        if not reached(v)
            reached(v) = true
            q.insert(v)
```

Time Complexity if using an adjacency list representation.

Suppose we are running the algorithm on a graph $G = (V, E)$. In the following we use the standard notation of $|V|$ for the number of nodes and $|E|$ for the number of edges.

For the inner loop, the time complexity is $O(degree(v))$ because in the adjacency list representation, the adjacency list for v contains exactly the nodes that are adjacent to v, and the body of the inner loop must execute for ash of them.

The number of time the outer loop executes depends on the queue q. Every node in the graph gets put on the queue once, and only once. So the outer loop will execute $|V|$ times. Each iteration of the outer loop requires $O(degree(v))$ for one of the nodes $v \in V$. If the nodes in the graph are $V = \{v_1, v_2, \ldots, v_n\}$ that means that one of the outer loop iterations will cost $O(degree(v_1))$, one of the outer loop iterations will cost $O(degree(v_2))$, one will cost $O(degree(v_3))$ and so on. Since $O(f(n)) + O(g(n)) = O(f(n) + g(n))$ we can write the total cost of the outer loop as:

$$O(degree(v_1) + degree(v_2) + \cdots + degree(v_n))$$

This means the time complexity depends on the sum of the vertex degrees. So what is the sum of the vertex degrees in a graph? Since each edge is incident on exactly 2 nodes, the sum of the vertex degrees must be twice the number of edges, or $2 \cdot |E|$. That means that the cost of the outer loop in the algorithm is $O(|E|)$.

Now we have to consider what comes before the while loop. There are a 3 statements plus and $O(|V|)$ for-loop. So the entire algorithm is $O(|E| + |V|)$ in the worst case.

Time Complexity if using an adjacency matrix representation.

If we are using an adjacency matrix instead, then the inner loop gets more costly. To find all of the adjacent nodes of v, we have to examine an entire row of the adjacency matrix, which costs $O(|V|)$ time. This means that every iteration of the outer loop will cost $O(|V|)$. The number of outer loop iterations is still $|V|$ in the case of the adjacency matrix, so we do the inner loop, which costs $O(|V|)$, $|\mathrm{V}|$ times, which is $O(|V|^2)$ overall. If we add in the initialization loop, we get an overall time complexity of $O(|V|^2 + |V|)$ or just $O(|V|^2)$.

So which is faster? $O(|V| + |E|)$ (adjacency list) or $O(|V|^2)$ (adjacency matrix)? To find out, we need to ask what is the largest possible value for $|E|$? If every possible edge is present (every node is directly connected to every other node) then there are $|V|^2$ edges and the breadth-first search on the adjacency list graph has time complexity $O(|V|^2)$, same as the adjacency matrix. But if the number of edges is significantly less than the maximum $|V|^2$, then the adjacency list representation will result in faster breadth-first searches.

Exercise 4

```
// An algorithm for depth-first
// traversal of a graph.
Algorithm dft(s)
s is the start node.

// V is the set of nodes in the graph
For each vertex v in V
    reached(v) = false

dftHelper(s);

// Recursive helper method for algorithm dft()
Algoirthm dftHelper(v)
v is a graph node

reached(v) = true

// perform the visit operation for v

For each node u adjacent to v
    if not reached(u)
        dftHelper(u)
```

Time Complexity if using an adjacency list representation.

For an adjacency list representation, the cost of a single recursive call to `dftHelper` is $O(degree(v))$ because we have to check very node in v's adjacency list to see if it has been visited yet. Since each node is visited only once, there are $|V|$ recursive calls, one for each node. If the nodes in the graph are $V = \{v_1, v_2, \ldots, v_n\}$, then the recursive call for v_1 costs $O(degree(v_1)$, the recursive call for v_2 costs $O(v_2)$ and so on. If we add up these costs for all nodes, then the total cost for all recursive calls is:

$$O(degree(v_1) + degree(v_2) + \cdots + degree(v_n))$$

which is the same expression we got for breadth-first search which we now know is $O(|E|)$. If we add in the for-loop in the initialization we get $O(|E| + |V|)$ for the overall time.

Time Complexity if using an adjacency list representation.

For the adjacency matrix representation, the cost of a single recursive call to `dftHelper` is now $O(|V|)$ because we have to examine an entire row of the adjacency matrix to find all of the nodes adjacent to v. The rest of the analysis does't change – we have $O(|V|)$ recursive calls, each of which costs $O(|V|)$, plus the $O(|V|)$ initialization, which amounts to $O(|V|^2 + |V|)$ or $O(|V|^2)$.

Once again the adjacency list representation will result in faster traversals when the number of edges is significantly less than the maximum.

Exercise 5

Breadth-first search

Modifications to the breadth-first traversal are shown in red.

```
// An algorithm for breadth first search
Algoirthm bfs(s)
s is the start node in the graph

q = new Queue()

For each vertex v of V
    reached(v) = false

reached(s) = true
q.insert(s)

while not q.isEmpty() do
    w = q.item()    // get top node on stack
    q.deleteItem()  // pop the stack

    if w is the sought vertex
      return w

    For each v adjacent to w do
        if not reached(v)
            reached(v) = true
            q.insert(v)
```

Depth-first Search

Modifications to the depth-first traversal are shown in red.

```
// An algorithm for depth-first
// search of a graph.
Algorithm dfs(s)
s is the start node.

// V is the set of nodes in the graph
For each vertex v in V
    reached(v) = false

foundVertex = null
dfsHelper(s);
return foundVertex
```

```
// Recursive helper method for algorithm dft()
Algoirthm dfsHelper(v)
v is a graph node

reached(v) = true

if v is the sought vertex
    foundVertex = v

For each node u adjacent to v while foundVertex == null
    if not reached(u)
        dfsHelper(u)
```

Exercise 6

Modifications from the solution to Exercise 5 are shown in red.

```
// An algorithm for depth-first
// search of a graph which returns the path to
// the sought vetrex.
Algorithm dfs(s)
s is the start node.

// V is the set of nodes in the graph
For each vertex v in V
    reached(v) = false

foundVertex = null
path = dfsHelper(s)
return path

// Recursive helper method for algorithm dft()
Algoirthm dfsHelper(v)
v is a graph node

path = null
reached(v) = true

if v is the sought vertex
    foundVertex = v
    path = v
    return path

For each node u adjacent to v
    if not reached(u)
        path = dfsHelper(u)
        if path != null
            path = v + path // add v to front of path
            return path

return path   // only occurs if sought node does not exist
```