Algorithms Homework 8

Liam Dillingham

November 19, 2018

1 Question 22.2-9

Let $G = (V, E)$ be a connected, undirected graph. Give an $\mathcal{O}(V + E)$-time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

After trying many examples on paper, I have come up with the idea that the best way to do this is to perform a depth-first search. For some edge $e \in E$ connected to our source node s, we follow a path from s to its neighbor v, each time checking two cases:

- Node v has no unvisited edges
- Node v is the source node s

Our code will work as follows: Follow an arbitrary path around the graph starting at node s, pushing each node onto the stack, and coloring it black. Once one of the above conditions becomes true, we stop and begin popping back through the stack, deleting edge from the adjacency structure and re-coloring the node white.

```
// all colors are initially assumed to be white
Let S be a new empty stack
Let s be an arbitrary node in G
s.color = BLACK
COMPUTE-PATH(G, s)

COMPUTE-PATH(G, s)
    for each vertex v in G.adj[s]
        if v.color != BLACK
            S.push(v)
            v.color = BLACK
            COMPUTE-PATH(G, v)
    // All neighbors are BLACK
    while S is not empty
        u = S.pop()
        u.color = WHITE
        Remove edge (u,v) // Depends on implementation
    Select another node w in G where G.ads[w] != 0
    COMPUTE-PATH(G, w)
// end
```

By popping nodes off of a stack, we are essentially visiting the nodes in opposite direction. This algorithm visits every edge and node twice, giving it a runtime of $\mathcal{O}(2E+2V) = \mathcal{O}(E+V)$

To escape a maze using pennies, we can simply leave a penny on the ground at any sort of vertex. A vertex is any point where a path ends, or where two paths meet, such as a 4-way, a tee, or an elbow. By doing this, we can ensure that we don't continue to revisit locations in the maze we have already been.

2 Question 22.3-7

Rewrite the procedure DFS, using a stack to elimate recursion.

```
DFS(G) // no recursion
    for each vertex u in G.V
        u.color = WHITE
        u.pi = NIL
    time = 0
    STACK is an empty stack
    for each vertex u in G.V
        if u.color == WHITE
            STACK.push(u)
            DFS-VISIT(G, u)

DFS-VISIT(G, u)
    while !STACK.isEmpty()
        v = STACK.pop()
        time = time + 1
        v.d = time
        for each w in G.Adj[v]
            if w.color == WHITE
                w.color = GREY
                w.pi = v
                STACK.push(w)
        time = time + 1
        v.f = time
// end
```

3 Question 22.3-10

Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G, together with its type. Show what modifications, if any, you need to make if G is undirected.

An imporant distinction that will help us determine the type of edge hooks on the color of a node during its transition from discovery to finish. It is worth noting that a vertex u is WHITE before time $u.d$, GRAY between $u.d$ and $u.f$, and BLACK thereafter. This means that if we encounter a WHITE node, it has yet to be discovered, and is therefore a tree edge. In addition, the book states that back edges are GRAY, and either forward or cross-edges are BLACK.

Also from the book, given two BLACK vertices u, v, and edge (u, v), this edge is a forward edge if $u.d < v.d$ and cross edge if $u.d > v.d$. With this information, we can construct our algorithm

```
DFS(G)
    for each vertex u in G.V
        u.color = WHITE
        u.pi = NIL
    time = 0
    for each vertex u in G.V
        if u.color == WHITE
            DFS-PRINT(G,u)

DFS-PRINT(G, u)
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v in G.Adj[u]
        if v.color == WHITE
            print("Tree Edge")
            v.pi = u
            DFS-PRINT(G, u)
        else if v.color == GRAY
            print("Back Edge")

        // Edge must be black
        else if u.d < v.d
            print("Forward edge")
        else
            print("Cross edge")
    u.color = BLACK
    time = time + 1
    u.f = time
// end
```

If the graph is undirected, then any node can access any node where there is an edge. Fromt Theorem 22.10, In a depth-first search of an undirected graph, every edge is either a tree edge or a back edge. Thus the second two cases in my pseudocode will simply never be executed, and they can be removed for simplicity, or left for generality.

4 Question 22.3-12

Show that we can use a depth-first search of an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v are in the same connected component.

Suppose we have an undirected graph G and a vertex $v \in G$, with k edges connected to other vertices. Then each vertex connected to v is a root of a subtree in the depth-first forest. Thus for each root connected to v will have a value $1...k$, and for some root i, all vertices connected will also share that value. The pseudocode for this is shown below:

```
CC-DFS(G)
    for each vertex u in G.V
        u.color = WHITE
        u.pi = NIL
    time = 0
    k = 1    // Initial k = 1
    for each vertex u in G.V
        if u.color == WHITE
            u.cc = k
            k = k + 1
            CC-DFS-VISIT(G, u)

CC-DFS-VISIT(G, u)
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v in G.Adj[u]
        v.cc = u.cc
        if v.color == WHITE
            v.pi = u
            CC-DFS-VISIT(G, v)
    u.color = BLACK
    time = time + 1
    u.f = time
//end
```

5 Question 24.1-3

Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all the vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges). Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

Note that due to the convergence property, if the path between two vertices u and v is equal to $\delta(u, v)$, then relaxing all edges between the two vertices will result in their path still being $\delta(u, v)$. That is, it will not change. We can exploit this property in our code by checking after each iteration to see if the path changes. This is where the $+1$ comes into our run time. Although it will only take m passes to compute the shortest path, we will need to try again and verify that there is no change. Thus it will take $m + 1$ iterations.

We can achieve this by using a sentinel flag, initializing it to TRUE, and using it as a control in a while loop. We modify our RELAX() method to return TRUE or FALSE if it can find a path shorter than the current one. The pseudocode containing the modified BELLMAN-FORD() and RELAX() methods is shown below:

```
BELLMAN-FORD(G, w, s)
    flag = TRUE // Flag set to check if the shortest path has changed
    INITIALIZE-SINGLE-SOURCE(G, s)

    while flag == TRUE
        flag = FALSE
        for each edge (u,v) in G.E
            flag = RELAX(u, v, w)

RELAX(u, v, w)
    if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.pi = u
        return TRUE
    return FALSE
// end
```

Observe that we can omit the TRUE/FALSE return values from the BELLMAN-FORD algorithm, as we are given a graph with no negative-weight cycles.