CSE 101 Homework 1 Solutions

Winter 2021

Question 1 (Clone Maze, 35 points)**.** *Steven's body has been split into two clones and he needs to reach the recombiner to fix it. Unfortunately, the clones still have only one mind between them and will always move in the same direction as each other. What is worse is that the laboratory that the clones are in is filled with traps!*

The laboratory is given as an $n \times n$ grid of squares. Some of these squares are marked as walls *and are impassible. Some are marked as* traps *and will kill Steven if a clone enters one. Two squares are marked as the two pads for the recombiner, and two squares are marked as the current locations of Steven's clones. When moving, Steven selects one of the four directions (up, down, left, right) and* both *clones move one square in that direction if possible (a clone does not move if there is a wall of the edge of the laboratory one square in that direction). If a clone moves into a trap square, Steven will die. Give an algorithm to determine if it is possible for Steven to get both of his clones to both of the recombiner pads. For full credit your algorithm should run in time polynomial in n.*

Solution 1.

The key idea in this problem is that we can treat the pair of positions of the two clones as a node of a graph, G. Then every action that Steven takes(move up/down/left/right) corresponds to an edge in this graph. If any clone enters in a trap point, then Steven can't move in that direction. If one of the clones hits a wall or the end of the maze, then that clone can't move in that direction, whereas the other clone can move, if possible. Therefore, first we define the $CheckValid$ function to check if the pair of points A, B are legal locations for the clones. Also, we use another auxillary function, MoveDirection(A,B,direction) which returns the next pair of points(node in G) that can be reached from (A,B). MoveDirection(A,B,direction) returns NULL if one of the clone moves into a trap point. If either of the clone moves into a wall or the end of the maze, then the MoveDirection(A,B,direction) function simply moves the other clone while keeping the position of this clone fixed. If no clone can move, then MoveDirection(A,B,direction) simply returns the current position.

```
CheckValid($A, B$):
    return $0 \leq A.x < n$ and $0 \leq B.x < n$ and $0 \leq A.y < n$ and $0 \leq B.y < n$ and $!A.trap()$ and
        $!B.trap()$ and $!A.wall()$ and $!B.wall()$
```

Now we can create the graph, G where a pair of points in the maze is a node in G and each node has at most 4 neighbors(depending) on how many neighbours are valid nodes in the graph.

```
CreateGraph():
    For point P1 in Maze:
        For point P2 in Maze:
            CurrNode=(P1,P2)
            if CheckValid(CurrNode):
                G.addNode(CurrNode)
                LeftNode=MoveDirection(P1,P2,left)
                RightNode=MoveDirection(P1,P2,right)
                UpNode=MoveDirection(P1,P2,up)
                DownNode=MoveDirection(P1,P2,down)
                if LeftNode!=NULL and LeftNode!=CurrNode:
                    G.addEdge(CurrNode $\longrightarrow$ LeftNode)
```

```
if RightNode!=NULL and RightNode!=CurrNode:
    G.addEdge(CurrNode ⟶ RightNode)
if UpNode!=NULL and UpNode!=CurrNode:
    G.addEdge(CurrNode ⟶ UpNode)
if DownNode!=NULL and DownNode!=CurrNode:
    G.addEdge(CurrNode ⟶ DownNode)
```

Now we can run Explore on this graph G starting from the initial positions of the two clones and with the positions of the recombiner nodes as the target. Since the total number of possible pairs of points in an $n \times n$ maze is n^4 and we visit each pair at most once, the running time complexity is clearly polynomial in n.

Question 2 (Proportional Connectivity, 35 points)**.** *The nation of Graphania exists on a small chain of islands. Recently, they have begun building (two-way) bridges connecting some of the islands. The government of Graphania wants to measure the progress of this new bridge system and to do so, they want to measure the probability that given two random Graphanian citizens, A and B that A will be able to reach B by travelling along the newly built bridges.*

You are given a list of Graphania's islands along with the fraction of the population living on each island, and which other islands each island has bridges to. Your goal is to compute the probability that a randomly selected pair of citizens can reach each other using these bridges. You can assume that once on an island, a citizen can reach anywhere else on that island.

(a) Give a linear time algorithm to solve this problem. [30 points]

(b) Does this algorithm work if some of the bridges are one-way instead of two way? If not, what goes wrong? [5 points]

Solution 2.

(a) The key insight here is that any pair of citizens can only reach each other if they are in the same connected component of islands. If we first compute the connected components of the islands, then the problem reduces to "Given citizens A and B, what is the probability that they are in the same connected component?" For each connected component C_j containing islands $x_{j_1}, x_{j_2}, ..., x_{j_{n_J}}$, let $p_{x_{j_k}}$ be the fraction of the population living on island x_{j_k} (this is given), and let $p_{C_j} = \sum_{k=1}^{n} p_{x_{j_k}}$ be the fraction of the population living in connected component C_j (this is just the sum of the fraction of the population living on each island in the component). Then we have $P(A, B \in C_j) = P(A \in C_j) \cdot P(B \in C_j) = p_{C_j} \cdot p_{C_j} = {p_{C_j}}^2$. Summing this up over all connected components yields $P(A, B \textit{ are connected}) = \sum_{j=1}^{n} P(A, B \in C_j) = \sum_{j=1}^{n} {p_{C_j}}^2$, which is our answer.

Here is pseudocode for our algorithm:

```
ProportionalConnectivity(G, island_pops):        // island_pops[v] = fraction of pop. on island v
    component_pops <- list of zeros, length |V|  // component_pops[c] stores fraction of
                                                 // pop. in component c
    ConnectedComponents(G)                       // run algorithm from class
    For v in G:
        component_pops[v.CC] += island_pops[v]
    total_prob <- 0
    For c in {1..component_pops.size}:           // iterate through components
        total_prob += component_pops[c]^2
    Return total_prob
```

We know from class that `ConnectedComponents(G)` runs in linear time. The rest of our algorithm involves twice performing a constant-time operation on (at most) each vertex of G. Thus, our algorithm is clearly linear time, as required.

(b) Note that in a directed graph, it is possible for A to be able to reach B even if A and B are not in the same strongly connected component (i.e. if B is unable to reach A due to some of the bridges on the path from A to B being one-way). Thus, the probability that A is able to reach B and the probability that A and B are in the same strongly connected component are no longer equal, and so our algorithm fails to work.

Question 3 (DFS Tree from Pre- and Post- Orders, 30 points)**.** *Given the pre- and post- order numbers of each vertex of a graph G in a particular execution of DFS, give a polynomial time algorithm that computes the DFS tree for that execution.*

Solution 3.

The key to solve this question is to understand the meaning of Pre and Post number of a vertex. The Pre number indicates when we start exploring a vertex and its neighbours. The Post number indicates when we finish exploring a vertex and its neighbours. Here is the pseudo-code for the algorithm:

This algorithm will try to find the all the children of vertex u in the DFS tree. if the $u.post == u.pre + 1$, then it means that when we applied the DFS algorithm on vertex u, all the neighbours of u have already been visited, which makes u a leaf. Otherwise, the vertex v such that $v.pre == u.pre + 1$ will be the next vertex to visit in the DFS algorithm. This makes v the first child of u. we add v to the children of u and then explore v until time $v.post$. After that we return to u to check whether all the neighbours of u have been explored, if not then the vertex w such that $w.pre = v.post + 1$ is the next children of u. We will continue until all the children of u has been found. If we repeat this algorithm for all the vertices in the Graph G, then we can construct the DFS tree.

```
// for each vertex u we want to find all the children of u
DFS_Tree(u, pre_list, post_list):
    if u.pre+1 == u.post:      // vertex u is a leaf
        return u.post+1
    next <- u.pre + 1          // pre_number of the first children of u
    while next != u.post:       // hasn't finish exploring u
        v <- vertex such v.pre == next // takes O(n) time
        add v to u's children
        next <- DFS_Tree(v,pre_list,pos_list)
    return u.post+1
```

The time complexity of of this algorithm is $\mathcal{O}(n^2)$ where n is the total number of vertices. Since for each vertex u in Graph G, we need to call DFS_Tree recursively to explore all the children of u. Let's denote $|u_c|$ as the number of children of vertex u in the DFS tree. Then the time complexity for DFS_Tree(u,...) will be $\mathcal{O}(n) * |u_c|$ since it takes O(n) to find vertex v such that $v.pre == next$. Since in a DFS tree, the number of edges is at most $n - 1$. So $\sum_u |u_c| = \mathcal{O}(n)$. As a result, the time complexity to reconstruct the DFS tree will be $\sum_u \mathcal{O}(n) * |u_c| = \mathcal{O}(n) * \sum_u |u_c| = \mathcal{O}(n^2)$. If we use a hash-map to store the pre/post numbers, then the time complexity can be optimized to $\mathcal{O}(n)$ since right now it takes $\mathcal{O}(1)$ to find a vertex with a specified pre number.