Homework 4 - Robert Duque

Monday, November 21, 2022 10:20 PM

Robert Duque - 1582182
I understand and agree to abide by the provisions in the University of Houston Undergraduate Academic Honesty Policy, This is my own work.

P0 (20pt) Prove that the following definitions are equivalent (assume undirected graph).

a) A tree is a connected acyclic graph.

b) A tree is a minimally connected graph; removing any edge disconnects the graph.

c) A tree is a graph that contains a unique path between each pair of vertices.

[hint: you can prove they are equivalent by proving a) ⇒ b) ⇒ c) ⇒ a). Or you can prove separately like a) ⇔ b) and a) ⇔ c).]

A) Given that a graph is acyclic, this means that there is only ONE path between two vertices else it would be a cyclic graph. An acyclic graph behaves as a tree because if we got rid of one edge, it would no longer be a graph, same goes for a tree. Therefore, a tree is a graph that is acyclic and connected

B) Same proof as the question above, a minimally connected graph is a graph to where the removal of one edge no longer makes the graph and viable connected graph. This is the same behavior as a tree, if we remove one edge, then the tree would no longer be a viable tree. Therefore, a graph is a tree if it is minimally connected.

C) If there is one unique path between each pair of vertices in a graph, this means that the graph CANNOT be cyclic since there can ONLY be one path. If this is the case, the graph is presented as a tree since trees operate in a similar way that there is only one path between 2 randomly selected vertices in the tree. Therefore, a tree is indeed a graph that contains unique paths between each pair of vertices.

P1 (20pt) Simulating Graph Algorithms. For the following graph G_1 on the left and G_2 on the right: (Node are labeled by letters, edges labeled by numbers).

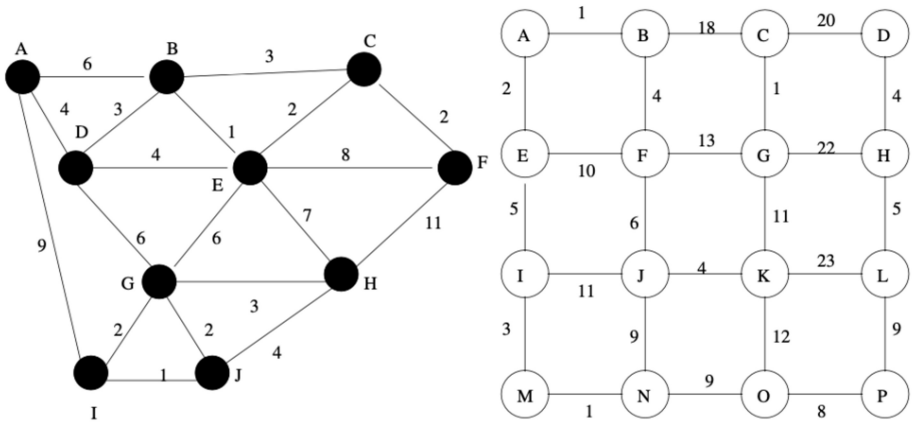


Figure 1. G_1 on the left and G_2 on the right.

a) Report the order of the vertices encountered on a breadth-first search starting from vertex A. Break all ties by picking the vertices in alphabetical order.

A breadth-first search is a queue type search to where the first vertex in the queue will be the first to get executed and finding vertices that have not already been searched in that specific vertex in the queue.
Below is the sequence of breadth-first search of G1:
A B D I C E G J F H

For G2:
A B E C F I D G J M H K N L O P

b) Report the order of the vertices encountered on a depth-first search starting from vertex A. Break all ties by picking the vertices in alphabetical order.

A depth-first search will use a stack data structure like algorithm that will push all connected vertices to the current vertex that is not already in the stack and traverse by popping the top of the stack.
Below is the sequence of depth-first search of G1:
A B C F H G J E D I

For G2:
A B C D H L K J I M N O P G F E

P2 (20pt) Given a pre-order and in-order traversal of a binary tree, is it possible to reconstruct the tree? If so, sketch an algorithm to do it; If not, give a counterexample.

Example: given pre-order traversal of binary tree: 1 2 3 4 5 6 7 8 9, and in-order traversal of the same binary tree: 3 2 5 4 6 1 8 7 9, the reconstructed binary tree is:

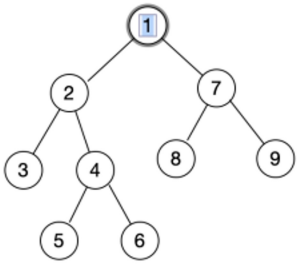


Figure 2. The reconstructed tree in the example.

[hint: try recursively building the left/right subtree]

It is, indeed, possible to reconstruct a binary tree given pre-order and in-order traversals of the binary tree. I will assume that the initial structure of the binary tree will be the pre-order traversal tree, therefore if the in-order traversal is different we would reconstruct the tree to represent the in-order traversal given.

We will assume that node class is pre defined to where each node has data related to it, a pointer to a right child and a pointer to a left child. We will make a function called Tree that will take both in-order and pre-order traversal arrays as arguments as well as a start index of the function array and end index of the recursive function array. In the main function, we will call the Tree function with in-order traversal, pre-order traversal, 0, and length of in-order traversal array - 1 for the parameters. The parameters will be called inOrder, preOrder, inBeg, inEnd , respectively.

Before we get into the function, we will initialize a global variable that will hold the current index of preOrder, we will call it preInd. In the function, if inBeg > inEnd, we return nothing which will end the recursive call to the function. If that condition isn't met, we will make a variable that will hold the node at preOrder[preInd]. We will then increment preInd by 1. We will check to see if the current node has any children nodes by checking to see if inBeg == inEnd, if it is return the current node. Else, we will create a new variable called inIndex that will use a search function that will search for the index of the current node in the inOrder array. We will then assign left child and right child of the current node by recursively calling the Tree function. We will then return the current node and this will reconstruct the tree.

Pseudocode:

```
preInd = 0
Def Tree(inOrder, preOrder, inBeg, inEnd):

    If(inBeg > inEnd):
        Return None

    Node = preOrder[preInd]
    preInd += 1

    If(inBeg == inEnd):
```

Return node

inIndex = search(inOrder, inBeg, inEnd, node->data)
^^^This function will basically use a for loop that goes in range(inBeg, inEnd) and will reutrn the current for loop index when the inOrder node data = node->data

Node.left = Tree(inOrder, preOrder, inBeg, inIndex-1)
Node.right = Tree(inOrder, preOrder, inIndex+1, inEnd)

Return Node

P3 (20pt) The *square* of a directed graph $G=(V,E)$ is the graph $G^2=(V,E^2)$ such that $u \to w \in E^2$ iff there exists $v \in V$ such that $u \to v \in E$ and $v \to w \in E$; i.e. there is a path of exactly two edges from u to v. Give efficient algorithms for both adjacency lists and matrices and analyze the time complexity.

G^2 will be a graph with edges that are associated with the edges of the graph G. An edge u->w in G^2 can only exist if there is a vertex v in G where u->v and v->w in graph G. The algorithm will be as follows:

Let's make the directed graph a binary graph to where if the edge exists, there is a 1 and a 0 otherwise. The algorithm will consist of a helper function called Edge that will take two vertices as arguments (v1, v2) which will essentially check if adjMat[v1][v2] == 1, if it exists return true, else return false. There will be a function called graphSquared that will take in the adjMat as a parameter and w as a that will hold all of the possible vertexes in adjMat. In the function, we will create a new adjacancy matrix that will be empty and the same size as adjMat (we will call it adjMat2). We will also create two for loops that will iterate through each cell of the adjMat. The first for loop will iterate with I and the second loop will iterate with j, in the nested for loop, there will be an if statement that checks if Edge(I,j) and Edge(j,w) exists in adjMat. If this statement is true, we add a 1 to adjMat2[i][w], else we will assign 0 to adjMat[i][w].

In the main function, we will call the graphSquared function using a for loop that goes through each possible vertex.

Pseduocode:

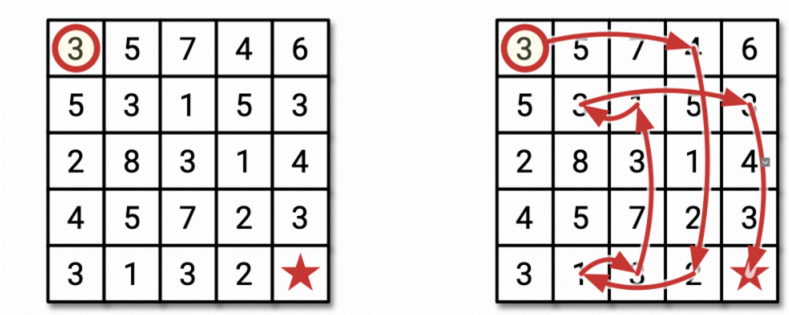
```
Def graphSquared(adjMat, adjMat2, w):
    For I in adjMat[0]:
        For j in adjMat[0]:
            If adjMat[i][j] == 1 && adjMat[j][w] == 1:
                adjMat2[i][w] =1
            Else:
                adjMat2[i][w] = 0

Initialize adjMat
Initialize empty adjMat2 (same size of adjMat)
Make a  vector V that has all vertexes in adjMat
For w in V:
    graphSquared(adjMat, adjMat2, w)
```

The run time complexity of this algorithm will turn out to be O(n^3). N being the amount of

vertexes there are in the graph, this because we have a for loop in the main function that will go through every vertex and in every iteration of that for loop, we will call the function graphSquare which contains a nested for loop. It would be a loop inside a loop inside a loop (loopception), therefore the runtime complexity will be n^3.

P4 (20pt) A *number maze* is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token out the edge of the board.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the number maze in the figure below, your algorithm should return the integer 8.



To solve this problem, we will look to the previous questions about breath first search (BFS) for a directed graph. Since this is a grid of numbers, we won't have to worry about edges of the graph but we will have to implement a new queue data type as well as another grid of the same size with booleans that will check if the specific cell has been visited or not.

We will start the algorithm by starting at the top left and take the number in the cell. Let's take the above example into consideration, we will implement a function that will take a cell that is up, down, left, or right how ever many units is in the current cell that we are in. In the example above, we can only go down or right 3 units since we are in the top left. We will put mat[0][3] into the queue as well as mat[3][0]. We will also implement a function that will use a while loop which will iterate through the queue until it is empty, when we go through each cell, we will mark it as visited (boolean) and do the same thing for that cell number with up, down, left, and right directions if possible and add to queue. We will do this until we reach the bottom right corner of the grid and we will also keep track of how many iterations there are in the traverse by assigning and integer along with each search in the queue. Let's take the example of going to mat[0][3] first, we will first insert it into the queue and when we execute that cell in the queue, we will put true for the visited cell as well as add 1 to an iteration number that we defined with each queue data. In this case, the above solution after getting executed after the queue would be ((mat[0][0], 0), (mat[0][3], 1), mat[4][2], 2), mat[4][1], 3), …) and so on.

Pseudocode:

```
Def BFS(mat, visited, ans, init = 0):
    Queue = []
    If init == 0:
        Queue.append(first element of mat, 0, 0, 0) # 2nd param is x, 3rd is y, 4th is iteration
    Else:
        Queue.append(init)

    While len(queue) != 0:
```

```
        If beginning of queue is equal to bottom left corner:
            Ans.append(queue[0][3])
        Else:
            If we can move to the left without going out of grid and visited == false for new cell:
                Queue.append(new cell int in mat, new cell x, new cell y, queue[0][3] + 1)
                Visited[new cell] = true

            If we can move to the right without going out of grid and visited == false for new cell:
                Queue.append(new cell int in mat, new cell x, new cell y, queue[0][3] + 1)
                Visited[new cell] = true

            If we can move up without going out of grid and visited == false for new cell:
                Queue.append(new cell int in mat, new cell x, new cell y, queue[0][3] + 1)
                Visited[new cell] = true

            If we can move down without going out of grid and visited == false for new cell:
                Queue.append(new cell int in mat, new cell x, new cell y, queue[0][3] + 1)
                Visited[new cell] = true

        Queue.pop(0)


Mat = any nxn grid
Visited = grid size of Mat with all cells = False
Ans = []
BFS(mat, visited, ans)
If(len(ans) == 0):
        Print("No Solution")
Else:
        Print(min(ans))
```

This algorithm will iterate through every possible combination of cells from the current cell and append this information to an queue that will hold every possible iteration. Each tuple will consist of the int of the cell, x of the cell, y of the cell, and the current iteration which will keep track of the smallest possible solution to the grid. The complexity of this algorithm will be O(n^2) since the algorithm will go through every possible iteration of the queue and check for neighboring cells.

Lets show some examples:

```
[[1 5 5 1]
 [2 5 2 2]
 [1 1 1 2]
 [1 1 2 0]]
9


[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 0]]
6


[[1 2 1 1]
```

```
[[1 2 1 1]
 [1 3 3 1]
 [1 1 2 1]
 [1 1 4 0]]
5

[[1 2 1 1]
 [4 3 3 3]
 [3 1 2 4]
 [1 3 4 0]]
No Solution

[[3 5 7 4 6]
 [5 3 1 5 3]
 [2 8 3 1 4]
 [4 5 7 2 3]
 [3 1 3 2 0]]
8
```