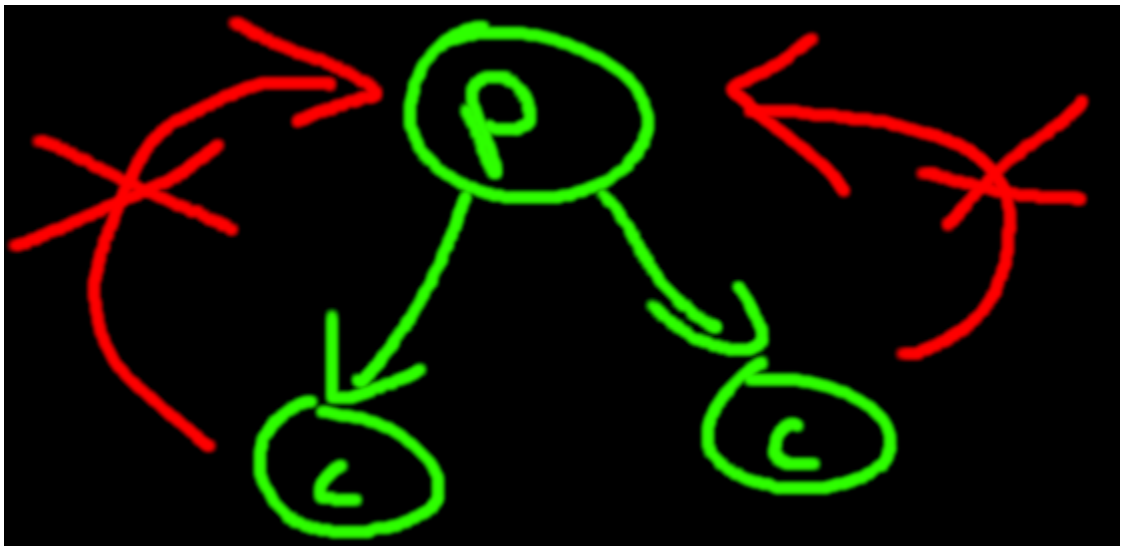
Jana Tahan
PSID: 1870644

P0

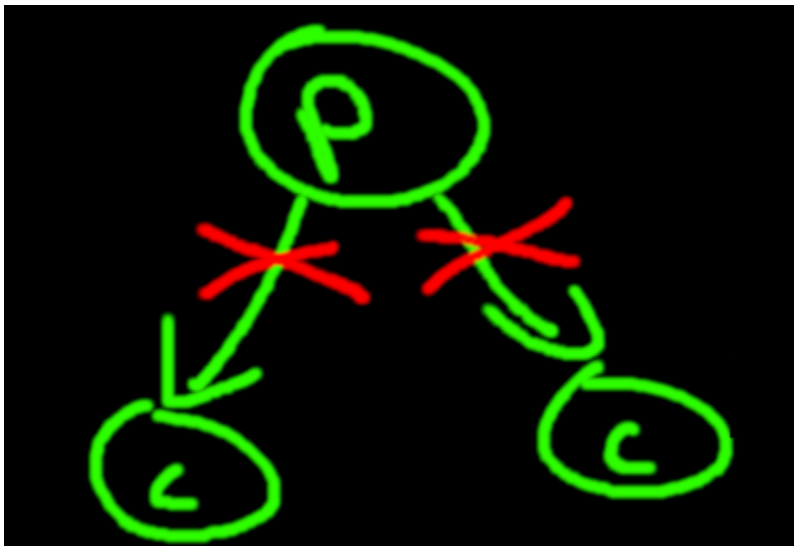
a) A tree is a connected acyclic graph

As nodes only point to THEIR children, the graph cannot form a cycle, also each node must have a parent node



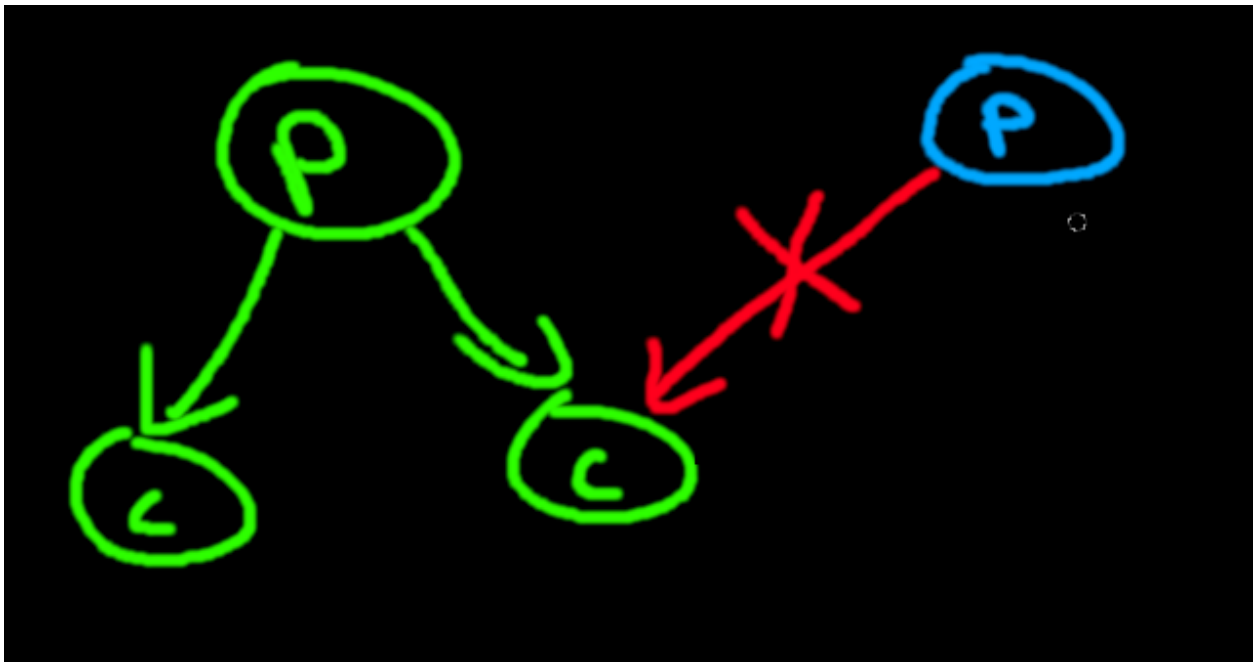
b) A tree is a minimally connected graph; removing any edge disconnects the graph.

As A states that a graph is connected, and C states that the connections are unique, removal of any edge (connection) will disconnect the graph as each node only has one connection back to the graph, and each connection represents a node's only connection to the graph.



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

As A states a tree is acyclic as each node only connects to its children, and B states that that edge is the only edge connecting that child node to the graph, then each vertex traversal would be unique.



P1

a)

G1) A D B I E G C J H F

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

b)

G1) A D B E C F H G I J

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

P2

```
typedef struct TreeNode {
   int label;
   TreeNode *left = nullptr, *right = nullptr;
} TreeNode;

using namespace std;

int dist (unsigned int root, int it, vector<int>& preorder){
   auto iter = find(preorder.begin() + root, preorder.end(), it);
   return distance(preorder.begin(), iter);
}

int minIdx (unsigned int root, unsigned int l, unsigned int r,
vector<int>& preorder, vector<int>& inorder){

   if(l == r)
       return -1;

   if(l + 1 == r)
       return l;

   int rootlocation = find(preorder.begin(), preorder.end(),
inorder[root]) - preorder.begin();

   int min = dist(rootlocation, inorder[l], preorder);

   int minI = l;

   for(l += 1; l < r; l++){
       int distance = dist(rootlocation, inorder[l], preorder);
```

```
        if(distance < min){
            min = distance;
            minI = l;
        }
    }

    return minI;
}

TreeNode* filltree (vector<int>& inorder, vector<int>& preorder, int i,
int l, int r){

    TreeNode* node = new TreeNode;

    node->label = inorder[i];

    int leftIdx = minIdx(i, l, i, preorder, inorder);
    int rightIdx = minIdx(i, i + 1, r, preorder, inorder);

    if(leftIdx != -1){
        node->left = filltree(inorder, preorder, leftIdx, l, i);
    }
    if(rightIdx != -1){
        node->right = filltree(inorder, preorder, rightIdx, i + 1, r);
    }

    return node;
}

void fillPostorder(vector<int>& preorder, vector<int>& inorder){

    int rootInd = distance(inorder.begin(), find(inorder.begin(),
inorder.end(), preorder[0]));

    TreeNode* root = filltree(inorder, preorder, rootInd, 0,
preorder.size());
}
```

I programmed this in c++ for programming challenge 3 for p5, it constructs a Binary Tree given preorder and inorder input, just as p5 had, in the problem we had to solve the postorder traversal, which I obtained by constructing the tree and then printing out a postorder traversal.

For proof of correctness this was run on the server with my PSID (listed above) and
Submission ID:
Adf7320b-9c5c-4c59-a268-79433de49ed3

Time complexity is O(n ^ 2) as the recurrence relation is T(n) = T(n - 1) + n;

P3)

Given an adjacency matrix we can check in constant time whether a given edge exists.

To discover whether there is an edge u->w for each possible intermediate vertex v we can check whether u->v and v->w exist in E

Since there are at most n intermediate vertices to check and n^2 pairs of vertices to ask about this takes O(n^3) time

With adjacency lists we have a list of all the edges in the graph

For a given edge u->v we can run through all the edges from v in O(n) time and put the results into the adjacency matrix G^2.

G^2 is an adjacency matrix of G which is initially empty It takes O(mn) to construct the edges and O(n) to init and read the adjacency matrix a total of O((n +m)n)
As n <= m unless the graph is not connected.

This usually results in an O(mn) time complexity and is faster than the previous algo.

P4)

Backtracking BFS is the best solution for solving this problem.

Make grid visited (n x n) of booleans. Visited[r][c] == 0 if unvisited and == 1 if visited

Make empty Queue “open” to represent squares that we are pushing to the BFS search.

```
def dist(a, b):
    r, c = a
    tr, tc = b

    d = abs((r - tr) + (c - tc))

    #print(d)
    return d

def backtrackBFS():
    open = Queue()
    visited = [[-1] * n for _ in range(n)]

    visited[n - 1][n - 1] = 0
    open.put([n - 1, n - 1]) #put bottom right item into queue to begin
backtracking

    while not open.empty():

        r, c = open.get()

        for dir in range(4):
            tr, tc = r, c
            while True:
                if dir == 0:
                    tr-=1
                elif dir == 1:
                    tr+=1
                elif dir == 2:
                    tc-=1
                elif dir == 3:
                    tc+=1
```

```
                if tr < 0 or tc < 0 or tr >= n or tc >= n:
                    break

                if visited[tr][tc] == -1 and dist([r, c], [tr, tc]) ==
grid[tr][tc]: #ignore if visited already as when a square becomes first
visited, it is the fastest path that square can reach the end.
                    visited[tr][tc] = visited[r][c] + 1

                    if tr == tr == 0: #we found our way back to the start!
                        return visited[tr][tc] + 1

                    open.put([tr, tc])
    return -1
```

This algo will return the length of the shortest path to reach the goal or -1 if not found with a recurrence relation of

T(n) = T(n - 1) + 2 sqrt(n - 1) + 1 with n being the number of squares, this results in a time complexity of

O(n log n)

as each square can be iterated upon and each iteration takes 2sqrt(n - 1) + 1 iterations which can be simplified to log (n) as log(n) = 2log(sqrt(x)) = 2(sqrt(n))