Question 1)

Algorithm Idea: I will base this algorithm on the BFS that Atri discussed in class. First, we will assume that all graphs we are dealing with are undirected. That means that the vertices that connect each nodes are able to move from one another using the same single line. We will create an empty array to store the nodes that we already went through. We then allow L to be our tree in a form of a linked list and let i be the nodes for the tree. i will be 0, since we are starting the root of the tree. We add the root of the tree in the array since we always start at the root of the tree. While the current node is not empty, we make sure the next node in the tree is not empty. Then, we check for any vertices that are connected to the node. If there is only one vertex, we must go through that path. If there are more than 1, algorithm chooses one by random. Add current node into the array. i will increase by one. Now, you check the array to see if there is any double nodes. If there is, this means there has been a cycle made by the algorithm. You break out of the loop and then print out the current array.

Algorithm Detail:

Let a be an empty array

Let L be a undirected tree in a form of a linked list

Set i = 0

add 0 to a

While Li is not empty

Li+1 is not empty

Check how many vertices are connected to current node.

If vertices == 1

Go through path

Else if vertices > 1

Randomly choose path

Add current node to a.

i++

Check if array has double node entry.

If a has double nodes

Break

Print (a)

Proof Idea: To prove the algorithm’s correctness, I will show that the algorithm produces a cycle at the end of the program. To compute that this algorithm runs in O(n+m) times, I will sum up the upper bounds of the running times for all steps in the algorithm.

Proof Correctness: According to the algorithm, we always start at node 0 since that is the root of the tree. Since we already have that node in the array, nothing more we can do. Once we search any available vertex connected to the root and move to the next node, we always add the node we just arrived at to the array. We check if the array has any double nodes. This allows us to check if any cycles happened. If so, that is what we want. For proving sakes, lets assume that the tree only has 3 nodes, a root and 2 another nodes. We start off with root. Root is added to the array. We check the vertex connected to root. Since root can go to either node 1 or node 2, we need to randomly choose one, but for proving sakes, we just travel to node 1. Node 1 is added to array. We check if there is any double nodes in the array. Since we only traveled once, there is no way we have double nodes at the moment. We loop back to the beginning and now check if node 1 has any vertex. Since this is an undirected tree, Node 0 and 1 can free travel between one another. Node 1 can go to either Node 0 or node 2. For proving sakes, we travel to node 2 and add node 2 to the array. We now check for double nodes in array. No doubles so far. Now we repeat again for node 2. Node 2 can go to either 1 or 0. Right now, I will give two different solutions that are both viable by algorithm. Node 2 can go to node 0. Node 0 is added to array. We check array for double nodes. Node 0 appears twice, 0 – 1 – 2 – 0. This is a cycle. Now, if we went to node 1 instead, we would get 0 – 1 – 2 – 1. This is still a cycle since 1 appeared twice. With these two different solutions, this proves that my algorithm is correct.

Run Time: Since my algorithm is heavily based on the BFS that was discussed in class, and we were able to prove that the BFS can run in O(n+m) times, I will refer to that algorithm as much as possible. The while loop in my algorithm only runs in m times since it’s the max number of pairs that it can produce. The if statements helps with that since they only check each vertex that is connected to the node. Also, near the end of the while loop, checking the array only runs in O(m) as well since only the nodes that we travelled to are added to the array. We may have multiple O(m), but the constants don’t affect much. So in the end, it ends up to be O(n) + O(m) = O(n+m).

Sources: Lecture notes 15

Question 2:

Algorithm Idea: We will have two linked list, one for group A and one for group B. We will have two variables, i and j and input n and list m where n and m aren’t equal to each other. i will be equal to 0 and j will be i + 1. We place i into group A. Now, we will have a for loop for every i less than n and j less than or equal to n. We check the current i and j and see if they are a pair from list m. If they aren’t a pair, we increase j by 1. If they are, we check the labels. If they are the same, we go to the group of node i. If node j isn’t create, we create one, then we make an edge to connect the two nodes. If the node is already created before, we connect the two nodes together. j increases by 1. Now, if the pairs are different, we go to the opposite group of i and check if node j is created in the other group. If so, we just increase j by 1. If it isn’t created, we create a node j and the other group and increase j by 1. If there is a label with neither same nor different, we can randomly place i and j in any group. j increases by 1. We repeat this process n times. However, once j reaches n, we increase i by 1, since n is the max we can do. This process repeats until every pairing in m is done. Once done, we now compare our linked list to m pairings. If they match, then it is consistent. If not, they aren’t consistent.

Algorithm Detail:

Let a be linked list for group A

Let b be linked list for group B

Set i = 0

Set j = i + 1

Add i to a

For every i less than n and every j less than or equal to n

We check if j equals to n

If j equals n

i increase by 1

j = i + 1

We check if the pair (i,j) exists in list m.

If (i,j) doesn’t exists

Increase j by 1

If (i,j) exists

Check label between (i,j)

If label is same

Check if j is in a

If j is in a

Add edge between i and j

If j is not in a

Add j to a

Add edge between i and j

J increase by 1

Else if label is different

Check if j is in b

if j is in b

j increase by 1

if j is not in b

add j to b

j increase by 1

Else if label is neither same or different

Randomly assign j to group

Check if j is either group

If j is in selected group

If j is in same group as i

Add edge between i and j

j increase by 1

if j is not in selected group

add j to group

If j is in same group as i

Add edge between i and j

j increase by 1

Compare a and b to m

If a and b match m

m is consistent

Proof Idea: To prove the algorithm correctness, I will show that my algorithm i) checks each label correct according to m ii) checks if m is consistent to what I found. To compute that this algorithm runs in O(n+m) times, I will sum up the upper bounds of the running times for all steps in the algorithm.

Proof Correctness: For i), I have if statements within the for loop to check each time we have a new (i,j) pairing. It check if that pair exists or not. If it doesn’t, it ignores that pair since there is no label between the two. Now, if the pair does exists, it checks if the label is either same, different or neither same or different. For same, puts them in the same group and adds an edge between the two. If different, they are put into different groups. If the label is neither, we random put j into a group since i is already in a group. For this part, we have to make sure the check if j is placed into the same group as i. If so, we have to have an edge between the two. This solves part i). For ii), we have to check if the listings of pairs m is consistent. Since we placed all the groups in two different linked list, we can easily check if the pairs in m are different or same based on the values are in the linked lists. For example, if pair (1,2) are the same, they should be in the same linked list, and if pair (1,3) are different, they should be in different linked list. If they are the same, it’s consistent. If not, they aren’t. This solves ii).

Run time: During the for loop, we have i run the full length of n, but j is n – i, since we can’t have j and i matching. Since checking if the pair exists in m or can takes, at worse can m times, we can assume that that runtime goes at O(m). Now, since each other if statements is only O(1), the sum of all that is O(1). Checking if node j exists in the linked list can take up to O(m) as well since each j found is exists in listing m. So at worse can, listing of m can be all same and the last pair can take up to m times. Since we have two of those checking, it becomes O(2m) but since constant don’t matter, it’s just O(m) Now, the for loop in total runs in O(n) times since i is running from the very beginning to the very end. If you take the sum of that for loop, it becomes O(n) + O(1) + O(m) = O(m + 1 + n) = O(m + n). Now, near the end of the algorithm, when checking if list a and b is consistent with m, we need to check everything, which can take O(m), so the run time for that is O(m). In the end, we get O(m + n) + O(m) = O(2m + n) = O(m + n).

Sources: Lectures notes.