Name: Jahan Kuruvilla Cherian

Discussion 1D
4:00pm – 6:00pm
TA: Arman

CS 180 – Homework 2

Exercise 7 – Page 28

We can make an analogy of this problem to that of a stable matching problem. We see this in the following capacity. A switching requires a perfect matching between an input and an output wire which involves choosing the correct match for data streams between the two. For the priority lists for the input and the output we define them as follows:

1.) For the input we wish to transfer the data stream as early as possible because this way it has less time to run the risk of running into another previously run data stream in the junction box. Thus we rank our *output wires in order of proximity from source to destination*.
2.) We apply the reverse for the output wires in that we want the data stream to be switched as late as possible for minimizing the risk of running into other data streams, and so we rank our *input wires in proximity from destination to source (reverse of source to destination)*.

Going with the definition of a *valid switching* as not having any run-ins of data streams at a junction box, we now want to solve the problem of finding the existence of an algorithm that provide the stable matching. As with any stable matching problem we have the following algorithm:

```
For port $I_i$ in all input wires
    $I_i$ transfers data to $O_j$ from its priority list
    if $O_j$ has not had data pass through it
        $O_j$ accepts and matches with $I_i$
    else
        if $I_i$ is higher in $O_j$'s priority list
            $O_j$ commits to $I_i$
            Previously matched input now has availability to transfer data with another output wire
        else
```

O_j remains committed to previous input wire

We can prove the stability of this algorithm using proof by contradiction. Let us assume run-ins between two data streams at a junction box meeting at input I_i and output O_j. One of these streams must have been from I_i and the other from a different input, say I_k onto the same output O_j. However, the jth output prefers the ith input because j meets i downstream from k, and likewise input i prefers output j over another output onto which it will be switched since it meets j upstream from said other output. This therefor contradicts the assumption of choosing a stable matching.

As with stable matching problems we represent the input and output wires as two lists with each element having a preference list, thus solving the stable matching algorithm in $O(n^2)$.

Exercise 5 – Page 108

Let us define the number of leaf nodes for some tree *T* as $n_L(T)$ and the number of nodes with 2 children in said Binary Tree as $n_2(T)$.
Base Case: Let the binary tree be a singular root node. In this case $n_2 = 0$ and $n_L = 1$ (the root itself). This satisfies the condition that $n_2 = n_L - 1$.
Hypothesis: Let us choose an arbitrary sub-tree within the Binary Tree *T* and call it *S*, and let it contain more than 1 node, where we define *L* as its leaf node. Because we state this sub-tree has more than one node and *L* is not the root, we know *S* has a parent *P*.
Inductive Step: Let us remove the leaf node thus forming a new sub-tree *S'*. If *P* had no more children, then we would have the case wherein $n_L(S) = n_L(S')$ and $n_2(S) = n_2(S')$. Using our hypothesis, we see the inductive step for this scenario satisfied, but let us look at the case wherein *P* has another child. In this case because of its nature of having two children, deleting one of the nodes would mean that the sub-tree still contains one child and thus we have $n_L(S) = n_L(S') - 1$and $n_2(S) = n_2(S') - 1$. By the principle of induction and the inductive hypothesis, we see that the relation is satisfied.

Exercise 7 – Page 108

To prove the claim that *G* is a connected graph if it has degree of at least *n/2* we use proof by contradiction. Let us assume *G* is not a connected graph. In this case we choose to look at one connected component of the graph and call it *C*. By the problem statement and seeing that in our assumption that there must now be at least 2 components, we can say that $|C| \leq n/2$. Looking at a single node *n* in *C*, we see that all of *n*'s neighbors must also be in *C* because otherwise this would not be disconnected. Thus we see that the degree of *n* is $|C| - 1 \leq \frac{n}{2} - 1 < n/2$. Thus we see that the degree of this node is not **at least n/2** since it would be less than that. Thus we have a contradiction which proves that *G* must be connected if every node is to have a degree of *n/2*.

Problem 4 from website

We notice that in order to get the fastest time, we need to send the two slowest people together. If we denote P_i as the person with *i* crossing time, then this means we have to send P_7 and P_{10} together at some point. The reason this is the case is because that way we basically knock out slow times together, while if we sent someone slow and fast together then we would have to add the second slowest time on the second run forward, which would increase the overall time necessary to cross the bridge.

After establishing the aforementioned fact we need to decide when to send the slowest pair over. We first think about sending them over as the first pair, but then we notice, that because we need to send one out of the pair back to carry the torch and guide the next pair forward, this would be a waste as we would have to send someone from the slow pair over twice when we can do better and send the slow pair over just once. This is achievable by sending the two fastest first, bringing one of the two fastest back, then sending the two slowest across the bridge, leaving us with someone already fast on the goal side to come back and grab the other fast person to walk back. The sequence can be described as follows:

Time Passed /minutes	People at start end of the bridge	Scheduling	People at the goal end of the bridge
0 minutes	P_1 P_2 P_7 P_{10}		
2 minutes	P_7 P_{10}	Send fastest pair across	P_1 P_2
3 minutes	P_1 P_7 P_{10}	Send one of the faster back with torch	P_2
13 minutes	P_1	Send slowest pair across	P_7 P_{10} P_2

15 minutes	P_1 P_2	Send the fastest from the goal back with torch	P_7 P_{10}
17 minutes		Send the fastest pair across	P_1 P_2 P_7 P_{10}

Thus the fastest time to get all 4 people across is **17 minutes**.

Problem 5 from website

The diameter of a binary tree can be defined as finding the maximum **height** between the left and right sub-tree and then comparing it to the longest path between a pair of nodes that passes through the current root. To calculate the maximum height of a binary tree we just run a *DFS* recursively by calling $\text{maxHeight} = \max(\text{getMaxHeight}(\text{left}) + 1, \text{getMaxHeight}(\text{right}) + 1)$ with the base case of a null node returning a height of 0.

To run this algorithm in O(n) time, we take notice of the invariant of a tree – that is that every node has only 1 in-degree. Because of this we run our DFS for each parent node only once. What we mean by this is that we recursively call the max on the diameter in a depth first manner trickling down to the leaf nodes only once. That is we only call this function once for every n node making the run time O(n). But for this to happen, we have to make sure to calculate the maximum diameter and height in one traversal by basically storing values in a *tuple*. The pseudo code would look something like below:

```
getDiameter given root:
    if the root is invalid return (0,0) //where first element is the height and the last is diameter
    left, right = getDiameter(root.left), getDiameter(root.right)
    height = max(left’s height, right’s height)
    diameter = max(left’s diameter, right’s diameter, (left’s height + right’s height + 1))
    return (height, diameter)
```

The reason we use the height is because it enables us to calculate the longest path between a pair of nodes through the vertex. This algorithm performs a DFS in one go from the root thus giving us a runtime of O(n).