Name: Jahan Kuruvilla Cherian

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

CS 180 – Homework 3

Exercise 4 – Page 107

From the problem statement we can realize that *n* specimens can act as our vertices and *m* judgments can act as our edges between two vertices that describe the kind of judgment made on the correlation between the two specimens. Thus from this idea we see the similarity to an undirected graph G = (V,E). We can construct this graph by creating nodes for each of the specimens and connecting them with edges based on the judgments made. The crux of the problem is to run through this graph and make sure all judgments are consistent in nature. Before we jump into the algorithm however we must take not of the fact that *G* need not be a connected graph since we might have two specimens have ambiguous judgment between them. Thus if we label each component of the graph as G_i then we must arbitrarily pick a starting node n_s. Within this component we run a BFS starting at the aforementioned node and for each node we visit that is already visited, we consider the judgment (that is the edge between some nodes n_i and n_j) and label these judgments as "same" or "different" based on the relation. After creating and labelling all the judgments we just loop through the graph again and make sure the judgments are consistent. The algorithm and its analysis are highlighted below:

```
For component in graph G:
    Assign starting node n and label it A
    Mark n as "visited"
    Initialize set S=n
    Define BFS Layer L(0) = s
    For i in layer:
        For node v in L(i):
            Look at each edge (v,u) incident to u
            If u not marked "visited":
                Mark u "visited"
                If judgment (v,u) was "same":
                    Label u same as v
                Else: #They are different
                        Label u as opposite to v
                Add u to S and layer L(i+1)
For each edge(v,u):
        If judgment == "same":
                If u and v have different labels:
                        Return Inconsistency
        Else: #judgment is different
                If u and v have different labels:
                        Return Inconsistency
```

Constructing a graph is $O(V + E) \to O(m + n)$, BFS runs in $O(m + n)$ and going through the judgments is $O(m)$ which leads to a final runtime complexity of $O(m + n)$.
We notice that in the problem running the BFS will create the consistency or inconsistency which will thus show the inconsistencies if they exist in the list of judgments. The BFS labelling is the only labelling other than inverting the labelling of each component of *G*. Therefore, if we find an inconsistency in this labeling the entire set of *m* judgments cannot be consistent by the definition of consistency in the problem, while if the labeling was all consistent with respect to the *m* judgments then we are done.

Exercise 11 – Page 110

Looking at the triples given in the problem statement (C_i, C_j, t_k) we notice the potential to represent the connectivity between computers as a directed graph *G* which we can represent as a hashmap of nodes where the key represents the node and the value is a list of all the connections and their timestamps. Scanning through the triples we can create vertices (C_i, t_k) and (C_j, t_K) with directed edges connecting them in either direction while also adding them to the hashmap described above. If the triple we encounter is not the first for either C_i or C_j then we include a directed edge from (C_i, t) to (C_j, t_k) {or with C_j if the latter was the case} where t is the previous timestamp. Using the properties of this hashmap we are able to construct nodes and edges in constant time per triple.

Since the problem asks to determine If a virus could have been introduced at C_a at time *x* and have infected computer C_b at time *y*, we should basically look through our lists until we get to the node (C_a, *x'*) where *x'* is <= x. From this point onwards we just need to traverse through the neighbors of C_a to find if the connection reaches C_b in time *y'* such that *y'* <= *y*. If said node is not reachable then we have no infection, else we do.

Let us prove this claim by first assuming that there is a path from (C_a, *x′*) to (C_b, *y′*). Through this we see the virus move between computers C_i and C_j at time t_k when said edges between nodes is traversed in the BFS. Basically as per our definition mentioned above, this would mean that the virus leaves C_a at a time less than or equal to *x* and reaches our destination C_b at some time less than or equal to *y* thus leading to an infection. Now if we had the converse situation wherein the virus leaves C_a at some time after *x* and arriving at C_b by *y* then we can build a path from (C_a, *x′*) to some point (C_a, *x′′*) wherein x lies in the range between [x′, x′′]. Each time the virus moves from some C_i to C_j at time t_k then we add an edge from (C_i, t_k) to (C_j, t_k) and then if it moves out of C_j at some time *t* after t_k then we add the sequence of edges in the path from (C_j, t_k) to (C_j, *t*). Thus when the virus arrives at the computer C_b at some time *y′′* we add the node (C_b, *y′′*) which by definition will mean the virus will have reached C_b in time since *y′′* <= *y*, thus completing the path. This thus proves that running the traversal from our aforementioned start point at the times gives based on the trace we can detect a virus infection.

We see we add $O(m)$ nodes and edges for all the triples in the trace data, which is constant time for the building of said graph (because of hashmap implementation) which thus takes $O(m)$ time for all m elements. The BFS will run in time linear in the size of the graph which also takes $O(m)$ which leads to our final runtime complexity as $O(m) \equiv O(m+n)\ where\ n = m.$

Exercise 7 – Page 191

From the problem definition of the processing of a job J_i we see that we have no control over the preprocessing time within the super computer as that runs independent of the ordering of jobs, and so what really matters are the finishing times of the jobs on the PC. Let us then base our Greedy Algorithm on the criteria of the finishing time of the last job on a PC. Thus we come up with an *optimal* schedule called *O* that will run the jobs in order of decreasing finishing time f_i.

To prove *O* is the most optimal solution let us assume another arbitrary schedule *S* that does not order on the basis of *O*. This schedule will contain two jobs J_i and J_k so that the latter runs after the former, but the finishing time for the first job is less than the time for the second → $f_i < f_k$. We can make this schedule better by swapping the order of these two jobs and let's call this new schedules *S′* (only swapped the two jobs mentioned earlier). Due to this swap we see that J_k now finishes before it would have done so in *S* since it is now scheduled to run earlier on a PC. Similarly J_i now schedules later, but because the super-computer hands off J_i to a PC in *S′* at the same time as it would have handed off J_k in S. Since $f_i < f_k$, job J_i finished earlier in *S'*. Thus our swapped schedule does not have a greater completion time.

This swapping basically reduces the pairs of jobs whose order in the schedule do not agree with the order of their finishing times without increasing the completion time. Applying a sequence of these swaps we see $S \rightarrow O$ based on the exchange argument given above without increasing completion time, and so because said completion time for *O* is not greater than that for any arbitrary schedule *S* we have shown that *O* is the most optimal solution.

Exercise 11 – Page 193

Arbitrarily label the edges in the graph *G* as $e_1, e_2, e_3, \ldots, e_m$ where the first *n-1* of these edges belong to *T*. If we let *d* be the minimum difference between any two differently weighted edges, then we would subtract di/n^3 from the weight of the edge *i* in order to create distinctive weights, and that the sorted order of the new weights are the same as some valid ordering of the original weights. From all the producible spanning trees of G, we see that T is the one whose total weight is reduced by the most, because of the originally sorted n-1 edges belonging to T. Thus it is now the unique MST of G and is the same tree that will be returned by Kruskal's algorithm.

Problem 5 from website

Given that we are given a DAG; we can first run topological sorting on this graph in *O(V+E)* time. Once we have the edges sorted we can safely say that they go from lower to higher. The only way we can have a Hamiltonian cycle in the DAG is if the pair of consecutive vertices are connected by an edge because we have to be able to go through all vertices without backtracking, and so in the ordered set of vertices for a Hamiltonian path to exist we need to have this continual paired connection.
Thus to see if the consecutive vertices are connected by an edge each we basically run through all V vertices in *O(V)* time thus leading to an overall runtime complexity of *O(V+E)*.

Problem 6 from website

Let us call the depth of U *d* and let us take an arbitrary vertex *u* on some level d of U. Because by its definition of layered nodes, we know that a BFS Tree from *v* indicates the shortest path from *v* to every node assuming each edge as equivalent. Thus there is no path in G of length less than d from *v* to *u*. If the depth of T were less than d, there would be a path in G of length less than d from *v* to *u*, given by the path in T, which is impossible as this would create the shortest path which would then be present in the BFS Tree U.