Name: Jahan Kuruvilla Cherian

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

CS 180 – Homework 6

Exercise 6 – Page 416

Because of the difference between the light fixtures and the switches, we can imagine creating a bipartite graph $G = (V, E)$ such that V is a partitioned into two mutually exclusive sets S and L where $s_i \in S$ is the ith switch, and $l_j \in L$ is the jth light fixture. Therefore there can only be an edge $e = (s_i, l_j) \in E$ if and only if there exists a line segment between s_i and l_j does not intersect any of the m walls in the floor plan. This can be determined using a plane sweeping algorithm in a greedy paradigm to check for intersections m times over. Thus using BFS we can create this bipartite graph G in $O(n^2m)$ time. We can then run a test in $O(n^3)$ time to see if we have a perfect/stable matching and determine if the floor is *ergonomic* if and only if G has said perfect matching, where we define a perfect match to have no intersections. By this definition we realize that the answer will be correct since such a pairing between the two sets of vertices will be such that there are no intersections and that all n switches and light fixtures will be matched.

Exercise 14 – Page 421

a.) We can start this problem off by creating a network flow graph by doing the following. We create a source C such that each node in X connects with the source with a unit capacity edge, and likewise we create a sink T with incoming directed edges from each node in S with capacity of $|X|$. Now if the $maxflow = |X|$ through the *Ford-Fulkerson* algorithm, then we see that there is an escape route, else one doesn't exist. This holds because if there is an escape route with $maxflow = |X|$ then there are unique paths from X to S so that $|X|$ units of flow get through to S where there are paths with $|X|$ capacity to the sink. Each node in X will have unit capacity coming in from the source C and because all the flow is going through to the final sink T and each edge between X and S are unit capacity, all of the flow must be using unique edges and thus we must have a unique path from every node in X to some node in S.

b.) Now that we are unable to share nodes, we have an issue with the generic *Ford-Fulkerson* algorithm. Essentially we might generate a set of paths that share the same node to give us the maxflow, but sharing nodes is prohibited, and so in reality we might not actually have an escape route. Another issue point is that if we simply run *Ford-Fulkerson*, find the maxflow, and then run a latter check to see if all nodes have unit flow out of them, then we might discard shared nodes, and thus say that we don't have a path, when in reality we might have an alternative path, that would still guarantee an escape route, but due to the failing of the aforementioned check, we would have said there exists no escape route.

So to make sure we still follow the constraints of the problem we run a modified *Ford-Fulkerson* in that we also give each node a capacity, and set this capacity to 1 (and also give the node an augmentation, so that we can backtrack). This is analogous to converting the shared node, into two nodes that are connected by an edge of unit flow capacity. This essentially makes it such that only one path will be allowed through this node, and thus cannot be shared. We then run the same algorithm as described in part a to determine whether any escape routes exist.

We can demonstrate a and b using an example. This example assumes that all the escape paths go through one special node v between X and S. Because v has the same in-degree and out-degree from different nodes, we have a bottleneck at v. This will satisfy the given algorithm in part a since we can use different edges, but part b will show that an escape route does not exist since it must all go through v and thus will no longer be available for further use, and thus the other populated nodes will not have an escape route.

Exercise 17 – Page 423

We can start by running *Ford-Fulkerson* to get all the unique paths that is part of the maxflow solution. This problem can be thought of through the min-cut of maxflow, trying to find the starting nodes from which we begin the cut between S and T, and then finding all nodes that are not part of the partition containing the nodes linked to the source. We can thus break this problem down into two phases.

The first phase will analyze each path to find the root of the cut. This can be done in $\log(n)$ time, using a Binary Search. We essentially start at the middle of the path at node v, $ping(v)$ and if it returns yes, then we can be confident in saying that the first half of the nodes are reachable and so we now consider the latter half of nodes in this path. If $ping(v)$ returned no, then we know the cut node exists in the first half and so focus on that. Thus we keep reducing the problem set by half, until we find the first

node where $ping(v)$ returns a no. We run this k times for all of the k unique paths from which there were deleted edges, thus having a total runtime of $O(klog(n))$.
The second phase is to find all the nodes that are not reachable. Now that we know the k sources from which the cut begins, we just run a standard BFS to find all the nodes that reach up to all the different k sources, and "subtract" those nodes from the overall set of nodes in the original network, to get all the nodes that are unreachable. This can be done in $O(|E|)$ (since we can assume the original network is connected), but has no relation to the number of pings.
Thus the overall algorithm, with both phases, runs in $O(klog(n) + |E|)$, but we only run $O\big(klog(n)\big)$ pings to find out the nodes from where the path is no longer reachable.

Exercise 19a – Page 425

Create a bi-partition of nodes, where one partition is k doctors, and the second partition is a list of the days in the week. The edges from some source S to each of the k doctor nodes, is of capacity $|L_i|$ and the edges from the days to some sink T are of capacity p_i, while the edges between the doctors and the days are of unit capacity. Now it is just a matter of running the a max-flow algorithm in polynomial time to find the paths such that the flow is equal to the sum of all $p_i's$. If we have such a path, then we must also check that the capacity of the edges into T are saturated. If the two conditions are satisfied, then we can return to each doctor the list of edges from said doctor to each of the days where the flow is non-zero (these are the days the doctor will work for). If the conditions are not satisfied, then we return there is no set of lists that satisfy the problem's constraints.

This algorithm runs in polynomial time, because finding the maxflow can be done in polynomial time (*Ford-Fulkerson* is pseudo-polynomial in $O(C|E|)$ but variations of the algorithm such as *Edmonds-Karp* can be run in $O(|V||E|^2)$ etc.), and the check for saturation is linear in the number of doctors there are. Thus the overall algorithm is also polynomial.

Problem 5 from website

This problem is in many ways very similar to the problem of finding the largest increasing subsequence (LIS). We note the following points, in that a box can only be placed on top of another if it's width and depth are smaller than the box below it, that we can rotate the boxes, and use multiple instances of the boxes which means we can have multiple rotations of a box as part of the maximum height stack.

We start by generating all rotations of the box such that the depth of the box is smaller than the width, thus leading to a possible of 3 rotations for each box. This can be stored in an auxiliary data structure (array) that will be 3 times the size of the original given list.

We then sort the boxes in decreasing order in terms of the base area $(d \bullet w)$. We sort by the base area, because if a box's base area is smaller than another's then it can potentially be stacked on top of the latter box. If it is greater, however, then the former box cannot be stacked on top of the later.

After this sorting we have a problem similar to (LIS) with the following set of state equations for optimality:

$$opt(i) = \max\big(opt(j)\big) + h(i), j < i, w(i) < w(j), d(i) < d(j)$$

We could also have $opt(i) = maximum\ possible\ stack\ heigh\ with\ box\ i\ at\ the\ top$ and if there is no j that satisfies the state equation above, then $opt(i) = h(i)$.

The maximum height will then be $\max\big(opt(i)\big), 0 < i < n$. This algorithm will thus run in $O(n^2)$ as we go through all n generations for i and for j. We have the following pseudo code:

```
def maxStackHeight(boxes):
    rotations = generateRotations(boxes) # Let this give an array of size 3n of all
valid rotations of all boxes in boxes array
    sort rotations in decreasing order in terms of base area
    opt = [x.height for all x in rotations]
    for i in range upto n:
        for j in range upto i:
            if rotations[i].width < rotations[j].width &&
              rotations[i].depth < rotations[j].depth &&
              opt(i) < opt(j) + rotations[i].height:
                  opt(i) = opt(j) + rotations[i].height
    return max(opt)
```