CMPS 102 — Winter Quarter 2017 – Homework 2

Christopher Hsiao - chhsiao@ucsc.edu - 1398305

Solution to Problem 1 - Box Picking

You are a box picker at SoftPillows and your job is to pick the boxes for the pillows your company is shipping out. The fit is rarely perfect, but if the box is too big you can stuff it with bubblewrap, and if the box is too small, you can compression the pillow. Specifically, if box b_j has length m_j and pillow a_i has length l_i. Given n boxes and n pillows each day, your job is to pick a box for each pillow, so that total cost is minimized.
One day your boss comes to you to let you know that they just bought the MTT BoxPicker3000, a machine that automatically matches n boxes to n pillows. It works by finding the pair (b_j, a_i) for which $|m_j - l_i|$ is minimum, assigning a_i to b_j, and repeating, until all pillows have been assigned. Being a proud descendent of a long line of box pickers, it falls to you to try to do better. Is the algorithm for the BoxPicker3000 optimal, or can you find a counterexample? Either way, describe an efficient optimal greedy algorithm for the problem and prove its optimality.

Algorithm. *Greedy: Pick boxes and pillows with the smallest size first. The resulting pairings formed by (b_i, a_i) where b_i and a_i are the i^{th} box and pillow respectively.*

Claim. *Any pairing that claims to be strictly more optimal than our greedy pairing must have adjacent inversions in either the boxes or pillows.*

Proof. As argued in class, any sorted array is a long chain of consecutive in-order pairs. If there exists a different solution set of pairs that is not the output of our greedy algorithm, then there must be adjacent elements out of order in either array. □

Now that we know for sure there must be adjacent inversions of some kind in a different solution set, let's define an inversion for this problem.

Definition. *An inversion in schedule S is a pair of boxes (or pillows) i and j such that $i < j$ but j is paired with the i^{th} pillow (or box).*

We will compare out solution set with the one that claims to be strictly more optimal than ours, i.e. has at least 1 less total cost. We scan both solution sets, and upon finding a disagreement in our pairings, we swap the inversion found at that pair.

Claim. *Swapping two adjacent, inverted jobs reduces the number of inversions by 1 and does not increase total cost.*

Proof. It is obvious that, given an adjacent inversion, swapping the two resolves the inversion, and thus reduces the total number of inversions by 1. The question is now: "How can we guarantee the resulting pairing does not increase total cost?"

First, recall our definition of an inversion in the schedule on the boxes array, for simplicities sake. We will define three "costs" to keep our eyes on. Let $C_{i,j,k}$ to represent these costs.

- C_k = Cost of all other pairs not in the inverted pair prior to swapping.
- C'_k = Cost of all other pairs not in the inverted pair after swapping.

- C_i = Cost of the left pair of boxes and pillows prior to swapping.
- C_i' = Cost of the left pair of boxes and pillows after swapping.
- C_j = Cost of the right pair of boxes and pillows prior to swapping.
- C_j' = Cost of the right pair of boxes and pillows after swapping.

Now, let's analyze what happens when we swap the i^{th} and j^{th} entries.

$C_k' = C_k$ Since cost is calculated based on (box, pillow) pairs, if no changes are made to those pairs, then the total cost between those pairs stays the same. Thus, $C_k = C_k'$.

$C_i' \leq C_i$ For the left pair, observe that the cost can only get lower. This is because any swaps made only replaces a larger box with a smaller one, otherwise the pair of boxes (b_i, b_j)wouldn't have been scrutinized as being an inverted pair. Thus, $[C_i' \leq C_i]$.

$C_j' \leq C_i$ It is less obvious why this is true. By definition, $C_j' = |m_j' - l_i|$, where the pair now has a new box length m_j' to subtract with the old pillow length l_i.

Claim 1. *The new cost C_j' could be worse than C_j, but it will never be worse than the old cost of pairing the box with the previous pillow, C_i.*

Proof. Since C_j' is the difference between the larger box with a larger pillow, regardless of what the new cost of this pairing is, it cannot be greater than the old cost was, since the old cost was calculated with the same larger box with a smaller pillow. □

Thus, any solution set of pairs claiming to be more optimal than ours can, through surgery (correcting inversions) become our greedy solution set of pairs, with at least no increase in total cost. However, since their solution set is equal to ours, they couldn't possibly have a better total cost, so by proof of contradiction, our greedy algorithm must be the most optimal.

□

TT BoxPicker3000

Claim. *TT BoxPicker3000 is not an optimal algorithm.*

Proof. With a pair of boxes $(8, 4)$ and a pair of pillows $(20, 7)$, BoxPicker3000 will choose the pairing $(8, 7)$ and $(4, 20)$, to yield a total cost of 17. However, the optimal pairing is $(8, 20)$ and $(4, 7)$, which yields a total cost of 15. □

Solution to Problem 2 - Parade Planning

You are scheduling a parade full for some **very** self-important groups, who have all submitted their float to the parade .Each group has a cutoff time, c_i, such that if their float comes on at time $t \leq c_i$, then all p_i members of the group will come to watch the parade. If the float comes on at time $t > c_i$, then all of them will stay home. All floats take exactly one minute to parade and can start as early as time 0: Your job is to maximize the number of people who show up. Give an efficient algorithm to schedule all floats given (c_i, p_i) for $1 \leq i \leq n$, where each $c_i \in \mathbb{R}^+$ and each $p_i \in \mathbb{N}$.

Algorithm. *Sort all groups by cutoff time c_i in descending order. Then, starting with the latest c_i, begin with $t = c_i$, and if there is any overlap at t, pick the group with the largest p_i. t decreases in value, and the algorithm runs until $t = 0$.*

This means that for every time t, the most number of people who can be present at that time are present at that time. We claim this to be the most optimal algorithm.

Claim. *Our algorithm yields an optimal scheduling of groups in the parade such that number of attendees is maximized.*

Proof. Proof by Contradiction. If someone claims to have a strictly more optimal (at least 1 more person showing up) than the output of our greedy algorithm, this means that we differ at some point in time t, where they have chosen a group with a strictly lower p_i than us at that time, since we always choose the largest possible p_i at any given time t. So, for any given point in time where they don't choose the group with the highest p_i, we simply choose the greatest p_i for any t. □

Of course, this works great except with one glaring question: What if that group has been scheduled to go earlier?

Claim. *If a group has been scheduled to go earlier than it is scheduled to go in the greedy schedule, you can always schedule it to go later so to match the greedy algorithms timing.*

Proof. If there is a conflict where a group was scheduled to go later in the greedy schedule, that means the group (c_i, p_i) can go at some later time t, at least when it goes in the greedy algorithm (it could still potentially go later, but the fact it doesn't means there's another group that will draw in a bigger crowd that has taken an even later spot). If it were to go at the time of the greedy algorithm, it would in fact at least keep the total number of people attending the event constant, or increase it. This is because at that time in the greedy algorithm, it is the group with the greatest p_i. □

What if a group simply isn't scheduled in the greedy schedule, but exists in the challenging schedule?

Claim. *If a group is scheduled in the challenging schedule and isn't in the greedy schedule while taking into account the previous conditions, then that group can simply be replaced by the group at that time s lot in the greedy schedule.*

Proof. If this group is scheduled and the time is truly the latest the group can go, that means that at that point in time it is not the group with the largest p. This means there is some other group (namely, the one chosen in the greedy algorithm), with the largest p out of all groups for that time slot. By replacing this old group with the group for that time slot in the greedy algorithm. As the group in the greedy algorithm must have some $p_j \geq$ the challenging group, so the total number of people can only stay constant or go up. □

This algorithm sorts the list of tuples (c_i, p_i) on c_i in descending order, This can be done in $O(nlogn)$ time using MergeSort. Then, in a single pass through the amount of time allotted, if we use a Max-Heap to return the group with the largest p_i, then finding the next max via Heapify takes $O(logn)$ This means the algorithm runs with an upper bound of $O(nlogn * logn)$ which gets absorbed into $O(nlogn)$.

Solution to Problem 3 - Graph of Spanning Trees

Let G be a connected graph with two different spanning trees, T and T'. We say that T and T' are neighbors if T contains exactly one edge that is not in T' and T' contains exactly one edge that is not in T.

Now, from any graph G, we can build a graph H as follows: the vertices of H are the spanning trees of G, and there is an edge between two vertices of H if the corresponding spanning trees are neighbors. Is it true that if G is connected, then H is connected?

Given a connected graph G, we can say by definition that it contains at least one spanning tree, since otherwise the graph wouldn't be connected. With this graph H, our goal is to return a path between u, v in H. In order to prove connectivity of H, it is enough to prove that we can always return such a path connecting two vertices in H.

To prove this, we will utilize properties of the cutset of a spanning tree.

Algorithm. *1. Given a pair of vertices (u, v) in the graph H, start at the source vertex u.*

2. Take a cutset of the spanning tree u of the graph G.

3. Ignore edges/vertices we've seen before.

4. Form some new edge e not originally in u to form a cycle.

5. Delete a original edge in u that spanned the cutset.

6. Store which edges were deleted and added.

7. You have now formed u'.

Claim. *u' is a spanning tree connected to u in the graph H by a single edge.*

Proof. In a cutset of a spanning tree, if you form a new edge across the cutset, you get a cycle. You can remove an edge in any cycle to get a spanning tree. Essentially, what we did was form a cycle over u, and then remove an old edge of u to get u', which is also a spanning tree, but now differs by u by exactly one edge. Namely, by the edge that was created and the edge that was deleted. As such, there must be an edge between u and u' in the graph H. □

Now we know that, given some vertex u, we are able to get to any neighbor of u using the algorithm above, simply by removing the edge that the neighbor doesn't have, and inserting the edge that the neighbor has, and that u doesn't.

Claim. *We can use this to form a path between any pair of vertices (u, v) in the graph H.*

Proof. Because the graph G is connected, all possible spanning trees of G overlap over some edges in some way. As such, this means that given any two spanning trees of G, we can transform them, one edge at a time, into the other. Every transformation is an edge in H, a combination of transformations forms a path in H connecting any two vertices (u, v) in H. □

To apply the algorithm to produce a path between two vertices (u, v) in the graph H has a runtime of $O(|V| + |E|)$. Since we never want to back-track in H, we make sure to never touch edges and vertices we've seen before. Thus, we really only touch all vertices and edges one time to traverse H. Therefore, our runtime is $O(|V| + |E|)$.

Solution to Problem 4 - Mixed Trees

Let $G = G(V, E)$ be an arbitrary, connected, undirected graph with vertex set V and edge set E. Assume that every edge in E represents either a road or a bridge. Give an efficient algorithm that takes an input G and decides whether there exists a spanning tree of G where the number of edges that represents roads is $\lfloor |V|/\sqrt{2} \rfloor$.

For this problem, $\lfloor |V|/\sqrt{2} \rfloor$ is essentially a red herring, as its value does not actually bear any significance with regard to the graph (as stated in class). Instead, our goal should be to prove that a graph with these properties exists. To do this, we need to find a range corresponding to the minimum number of roads present in any vertex of H, and the maximum number of roads present in any vertex of H. We can apply Prim's algorithm to help us find these numbers.

Algorithm. *1. Assign weights of road = 1, and bridges = 0.*

2. *Run Prim's algorithm on G and get a spanning tree with the minimum number of roads and most bridges, with total cost* $weight_{roads}$
3. *Set* $r_{min} = weight_{roads}$
4. *Assign weights of road = 0, and bridges = 1.*
5. *Run Prim's algorithm on G and get a spanning tree with the minimum number of bridges, and the most roads, with total cost* $weight_{bridges}$
6. *Set* $r_{max} = (v-1) - weight_{bridges}$.
7. *If* $r_{min} \leq \lfloor |V|/\sqrt{2} \rfloor \leq r_{max}$*, then a spanning tree with* $\lfloor |V|/\sqrt{2} \rfloor$ *roads exists. Otherwise, it doesn't exist.*

Claim. *We can find the minimum number of roads* r_{min} *needed in a spanning tree.*

Proof. Since Prim's is designed to minimize total weight, By assigning all bridges a weight of 0 and roads a weight of 1, it stands that the spanning tree output by Prim's will take as many bridges as possible, unless it is forced to take a road (i.e. at some vertex u, the only available paths from u are roads). So with the output of Prim's, we get some spanning tree, and more importantly some $weightroads$, which equals r_{min}. □

Claim. *We can find the maximum number of roads* r_{max} *in a spanning tree.*

Proof. We use Prim's again, except now we set the weights of all bridges to 1, and the weight of all roads to 0. Thus, Prim's will produce an MST with the least amount of bridges, since it only takes bridges when forced to. Thus, we are left with a spanning tree with the maximum number of roads possible in a spanning tree of G. There can't be any more roads because adding a single edge in a spanning tree forms a cycle. Keep in mind that spanning trees have a constant number of edges $|V| - 1$. Now, taking the total cost output $weightbridges$, we can calculate the total number of roads in the spanning tree by subtracting the number of edges in total by the number of bridges $(v-1) - weight_{bridges}$. □

Now if we know $r_{min} and r_{max}$, we can safely say that if there exists a spanning tree with $\lfloor |V|/\sqrt{2} \rfloor$ roads, then it must satisfy $r_{min} \leq \lfloor |V|/\sqrt{2} \rfloor \leq r_{max}$.

The runtime for Prim's is $O(mlogn)$, and since our algorithm is simply applying Prim's algorithm twice and doing constant time arithmetic operations on the outputs of both executions of Prim's, then comparing the values with $\lfloor |V|/\sqrt{2} \rfloor$, the algorithm runs in $O(mlogn)$.