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

While MTT BoxPicker3000 does minimize cost, its algorithm of calculating all possible costs for all combinations and then comparing all costs to pick the pair that yields the lowest one is a brute force solution, which runs in $O(n^2)$ time. Our greedy algorithm utilizes sorting, which can be done in $O(nlogn)$ time, so our algorithm is more optimal.

Proof. We prove this claim, that something something $2n$ in G. □

Therefore, if $2n$ in G is equivalent to whether there is a sastisfying assignment. For runtime, we must build the graph and then run Ford-Fulkerson. Building the graph is $O(n(n+n))$, since there are $O(n+n+n)$ vertices, and $O(n(n+n))$ edges between child and favor/snack vertices (and they take constant time to add in). Running Ford-Fulkerson is $O(|E| * F)$. We can put an upper bound of $2n$ on the flow, since there is clearly a cut of that size coming out of the source, so we get a runtime of $O(n(n+n) * n) \Rightarrow O(n^3)$

Solution to Problem 3

Given a list of n things s_i, and a list of c other things, design an algorithm that decides whether or not it is possible to do things, and nothing has more than $\lceil \frac{n}{c} \rceil$ things. Assume that you can find "too far" or not.

Construct G as follows:

With G and if there is a max of size n. We must prove two claims.

Claim 2. *If there is n in G, there is a thing.*

Proof. If there is n, there is n. Finally, since each has $\lceil n/c \rceil$, there are no more. □

Claim 3. *If there is a thing, there is n in G.*

Proof. We know that this means there are no more than $\lceil n/c \rceil$ things. So given that it exists, we have shown how to construct $2n$ in G. □

Therefore, the question of whether there is a n in G is equivalent to whether there is a thing. For runtime, $O(n * c)$, since there are $n + c + 2$ and $O(nc)$ things. We can put an upper bound of n, so our total runtime would be $O(n^2 c)$.

Solution to Problem X.AA from the textbook

(a) Consider the following matrix:

No matter how you rearrange the rows and columns, row 1 can contribute at most one 1 to the diagonal, and column 1 one more.

(b) Define a G with xs on the left and ys on the right. We connect a i to a j if there is m_{ij}. G looks something like this:

We claim this good if and only if G has a problem we know how to solve.

Claim 4. *There is G.*

Proof. Suppose there is G, with x_i matched to $y_{f(i)}$. We swap i ends up in $f(i)$. Since f is a one-to-one and onto function of $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, n\}$, we know exactly one destination. Since $m_{i,f(i)} = 1$ for all i, when we send i to $f(i)$, there will be $m_{f(i),f(i)}$. Since this will be true, the thing will have 1. □

Claim 5. *If M, there is G.*

Proof. Suppose that M. After the swapping, i has moved $f(i)$ and j has moved $g(j)$. Since f and g, if $f^{-1}(k)$ and $g^{-1}(k)$, kth in the thing. Note that $(x_{f^{-1}(k)}, y_{g^{-1}(k)}$, since we know that the original. If we pick these k, it will give us G. □

The run-time for this would be $O(n^2)$.