CS 38, Final

Codename: Jambul

June 9, 2023

1 Problem 1

Given a function $f(x)$ defined over the integers $1, ..., n$, we seek the function $g(x) = ax + b$ that satisfies

$$\min \sum_{k=1}^{n} |g(k) - f(k)|,$$

for $a, b \in \mathbb{R}$.

From lecture, we know that a linear programming problem is any optimization problem in which both the optimization function and the constraints are linear. Denote $e_i = |g(i) - f(i)|, e = (e_1, ..., e_n)$. Then our problem is

$$\min \sum_{k=1}^{n} e_k, \text{ s.t. } e_k = |g(k) - f(k)|.$$

Note that, since we wish to minimize this summation, we can exchange our constraint for the seemingly more relaxed one, $e_k \geq |g(k) - f(k)|$. This constraint is then equivalent to the requirement that

$$-e_k \leq ak + b - f(k) \leq e_k,$$

so we can write the problem as

$$\min \sum_{k=1}^{n} e_k,$$

$$\text{s.t.} - e_k \leq ak + b - f(k),$$

$$ak + b - f(k) \leq e_k,$$

for $a, b \in \mathbb{R}$. Then we can see that this is an optimization (a minimization) problem with a linear optimization function (linear in e_k) and with linear constraints. Thus we have cast the original problem as a linear programming problem.

2 Problem 2

Algorithm 1. (Bipartite Checking.) This algorithm takes as input some connected, undirected graph $G = (V, E)$.

1. Create a queue `queue` that initially holds some arbitrary vertex v in the graph.
2. Color v red. All other vertices in G are initially uncolored.
3. Create an initially empty set `processed`.
4. While $|\texttt{queue}| > 0$:
   (a) Pop `curr` from `queue`.
   (b) For every neighbor `nbr` of `curr`:
      i. If `nbr` is in `processed`: Continue.
      ii. Else if `nbr` and `curr` are the same color: Return `False`.
      iii. Else if `curr` is red: Color `nbr` blue and add it to `queue`.
      iv. Else if `curr` is blue: Color `nbr` red and add it to `queue`.
   (c) Add `curr` to `processed`.
5. Return `True`.

Algorithm 2. (Directed Cycle of Odd Length.) Let our input be a directed graph $G = (V, E)$.

1. Decompose G into a set S of its strongly connected components using the SCC decomposition algorithm presented in class.
2. Create an empty set A.
3. For $s \in S$:
   (a) Convert each edge e in s into an undirected edge.
   (b) Add this undirected version of s to A.
4. For $a \in A$:
   (a) Run "Bipartite Checking" on a. If the output is `False`, return `True`.
5. Return `False`.

Proof of correctness. The algorithm presented above first decomposes G into its SCCs. For each SCC s, it converts all edges in s into undirected edges and checks if this undirected component s is non-bipartite; if it is, then the algorithm returns that there exists an odd-length cycle in G.

It is easy to verify the correctness of the "Bipartite Checking" algorithm. From definition, an undirected graph is bipartite if and only if it is 2-colorable. Thus we simply check if the given undirected SCC is 2-colorable or not by trying to 2-color it. The correctness of our algorithm therefore hinges on the claim that a graph G having an odd-length cycle is equivalent to G having at least one (undirected) SCC that is non-bipartite.

($\implies$) Suppose G has an odd-length cycle C. Then C is strongly connected, so this cycle must exist within some SCC of G. Let this SCC be denoted s. Then clearly if we convert all edges in s to undirected edges, this odd-length cycle will still exist in the undirected version of s.

We know from elementary graph theory that an undirected graph will be bipartite if and only if it has no cycles of odd length. Thus if the undirected component s has a cycle of odd length, then it will be non-bipartite, and thus G will have at least one undirected SCC that is non-bipartite.

($\impliedby$) Now suppose that G has at least one undirected SCC s that is non-bipartite. Then there must be a cycle C of odd length in this undirected SCC. Suppose now that C is not a directed cycle. Then, taking the directed version of C, where the directions of the edges come from the original, directed SCC, there must be some edges that point in the clockwise direction and others that point against this direction.

Suppose, WLOG, there are more "clockwise" edges than "counter-clockwise" edges. Take any such edge $e = (u, v)$ that points in the clockwise direction. Then, since s is an SCC, v must have some path to u in the "counter-clockwise" direction. If this path is of odd length, we may simply replace e by it in C to get an odd-length cycle. Otherwise, if it is of even length, then we can take it and add e to get an odd-length cycle.

Complexity. We know from lecture that decomposing G into its SCCs can be done in $\mathcal{O}(n + m)$ time. Then, the "Bipartite Checking" algorithm will run a modified version of BFS on each SCC in $\mathcal{O}(n_i + m_i)$ time, for n_i, m_i the number of vertices and edges in the given SCC, respectively. Thus the final loop in Algorithm 2 will, at worst, loop over all vertices and edges once, resulting in a runtime of $\mathcal{O}(n + m + n + m) = \mathcal{O}(n + m)$, as desired.

3 Problem 3

Algorithm 3. (k-Center Ultrametric Clustering.) Let `pairwise` be an $n \times n$ matrix such that `pairwise`(x, y) holds the distance from point x to y in the ultrametric space. We will use a variant of Kruskal's MST algorithm to solve the problem. Let k be the desired number of clusters.

1. Create a weighted, undirected, complete graph $G = (V, E, w)$ with $V = \{1, ..., n\}$ and $w(x, y) =$ `pairwise`(x, y) for $(x, y) \in E$.[1]
2. Sort the edges in E by minimum weight into a set S.

[1] This is effectively a copy of K_n with weights taken from the input matrix.

3. Create a forest F where each vertex in the graph is a separate tree.
4. While S is not empty and $|F| > k$:
   (a) Remove the edge of minimum weight from S; call this edge e.
   (b) If e connects two separate trees in F, then add e to combine those two trees into one.
5. Return F.

Proof of correctness. The algorithm above first represents the points in the ultrametric as a complete graph with edge weights given by the distance between those points in the ultrametric. It then uses a variation of Kruskal's algorithm for MSTs to construct the k clusters from the data.

The algorithm initially considers each individual vertex as its own tree in a forest. It then progressively considers the edge of minimum weight from the graph and, if that edge connects two separate trees in the forest (and does not induce a cycle), it includes it in the forest, merging the trees.

Checking the correctness of the algorithm therefore hinges on checking the claim that this greedy approach yields a correct clustering of the points. We note first that the constraint that, for any points u, v, w in the space,

$$d(u, v) \leq \max\{d(u, w), d(w, v)\}$$

necessarily implies that all triangles formed between points in the space must be acute isosceles triangles. This observation will serve to confirm the correctness of our algorithm.

Note that, assuming we do not yet have k clusters in our forest (otherwise the algorithm would have terminated), our algorithm always chooses to add the edge e of minimum weight to the forest so long as e does not induce a cycle in the forest. In this way, we effectively ensure that our final clusters $C_1, ..., C_k$ are chosen so as to be spaced as far apart as possible, since we continually opt to exclude the edges of largest weight (distance) for as long as possible.

Then, we know that in our ultrametric, all triangles between points must be acute isosceles. Because of this fact, it must be the case that, for $v, w \in C_i, 1 \leq i \leq k$, and $z \in C_j, i \neq j$,

$$d(v, w) \leq \max\{d(v, z), d(z, w)\}.$$

We note that the edges $(v, w), (v, z), (z, w)$ will always be present in G, since the graph is complete. In other words, then, in the (acute isosceles) triangle Δvwz, the edge (v, w) will be shorter than or equal to (w, z) and (v, z). The edge between these points (and thus all others in this cluster) will indeed be preferable to include in the cluster over any other edge excluded (which would then be connecting clusters).

Thus we can be certain that, for all final clusters, the edges included in the cluster will minimize the maximal distance between points better than any edges that were excluded from the clusters.

Complexity. Building the graph representation of the points takes $\mathcal{O}(n^2)$ time, as we iterate over half the cells in the input matrix. Since the graph is complete, there are exactly $m = \binom{n}{2} = \frac{n(n-1)}{2} \in \mathcal{O}(n^2)$ edges in this graph. Sorting these edges will take $\mathcal{O}(m \log m) = \mathcal{O}(m \log n) = \mathcal{O}\left(\frac{n^2-n}{2} \log n\right) = \mathcal{O}(n^2 \log n)$. The `while` loop will take $\mathcal{O}(n)$ iterations to terminate. Each iteration takes $\mathcal{O}(\log n)$ time to pop the min weight edge from S and $\mathcal{O}(n)$ time to naively check if e combines trees in F.

Thus the runtime of the algorithm is dominated by the sorting step, which is done in $\mathcal{O}(n^2 \log n)$ and is indeed polynomial (a looser bound on the number of steps is $\mathcal{O}(n^3)$).[2]

4 Problem 4

4.1 Part a

Let $G = (V, E)$ be an undirected, weighted graph.

Proof. ($\implies$) Suppose T is an MST of G for weights w. Then, from a past problem set, we know that any other MST T^* of G must have the same sorted list of edge weights. That is, T and T^* must have the same edge weights in the same frequencies; they may only differ in the particular edges they include.

Thus it must be the case that if we transform all edge weights in G by $f(w) : \mathbb{R}_+ \to \mathbb{R}_+$, then the sorted edge weights of T will again align with T^*. It remains to be shown that any other subtree H of the original graph that was not an MST would not be able to have a lower total edge weight than T for f.

This can be seen by noting that, if H was not an MST of the original graph, then the sorted list of edge weights for H must have that each entry either maps to an entry of the same weight in T's list or is greater than all entries in T's list. This is because, otherwise, if H had an edge of lower weight than all in T's list, then this edge would have been included in T when following Kruskal's algorithm, and from the logic above, this would imply all MSTs of G would have this edge in their sorted lists of edge weights.

Thus when transforming the weights in H's list by $f(w)$, it must be the case that at least one transformed weight will be greater than all other transformed weights in T, since f is strictly increasing. All other transformed weights in H's list will map to some particular entry in T's list of transformed weights (since none can map to a value smaller than all those in T's transformed list). This implies that the sum of the transformed weights in H must be greater than that for T. Thus indeed T maintains its status as an MST of G for weights $f(w)$.

($\impliedby$) Now suppose T is an MST of G for $f(w)$. From functional analysis, we know that if $f(w)$ is a continuous, strictly increasing function from $\mathbb{R}_+$ to

[2] If we were to store the edge weights in a min heap instead and pop from that, we could build the heap in $\mathcal{O}(n)$ time, making the loop the dominating step, and giving the algorithm an overall runtime $\mathcal{O}(n^2)$.

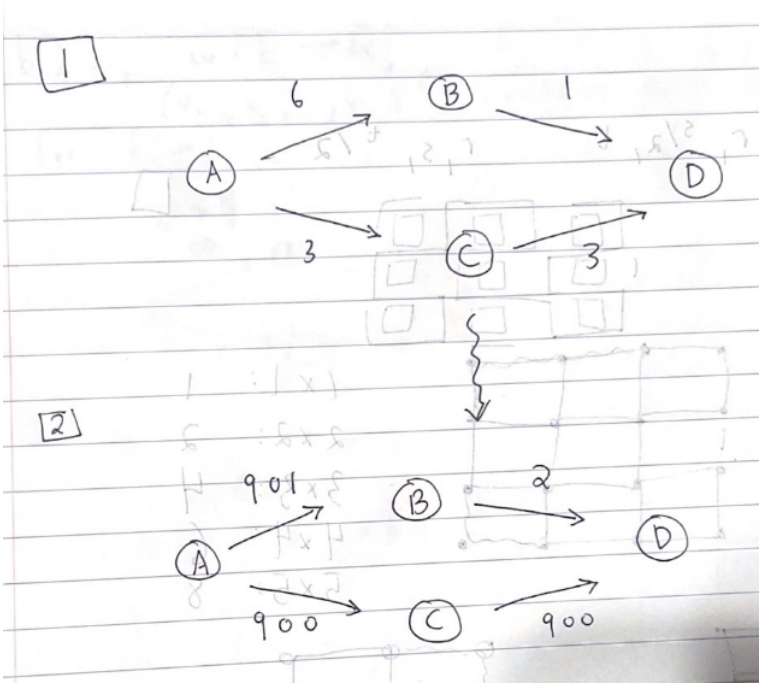
$\mathbb{R}_+$, its inverse f^{-1} must be strictly increasing from $\mathbb{R}_+$ to $\mathbb{R}_+$.[3]

Then we can use the logic above for the forward direction to confirm that any tree T that is an MST of G for weights $f(w)$ will be an MST of G for weights w, as we can reuse the same argument after exchanging w for $f(w)$ and $f(w)$ for f^{-1}. This completes the proof of the biconditional.

□

4.2 Part b

FALSE. The picture below demonstrates an example graph with an example $f(w)$ for which the theorem does not hold.



In particular, we can see that the graph is $G = (V, E, w)$, with $V = \{A, B, C, D\}, E = \{(A, B), (B, D), (A, C), (C, D)\}$, and $w(A, B) = 6, w(B, D) = 1, w(A, C) = 3, w(C, D) = 3$. The shortest path from A to D is clearly found by traversing the nodes in the order $A \to C \to D$, netting a weight of 6.

However, after transformation by $f(w)$, we see that $f(1) = 2, f(3) = 900$, and $f(5) = 901$. This function is indeed strictly increasing for these values (and a particular instance of f can be found by taking the linear interpolation of these values, which will be piecewise-continuous). Now, the path $A \to C \to D$ is no longer optimal, as this path will cost a total 1800, as opposed to 903 from

[3]The proof of this fact is relatively brief: If f is strictly increasing, then $f(x) < f(y) \iff x < y$, so assuming f^{-1} exists, $f^{-1}(x) < f^{-1}(y) \iff f^{-1}(f(x)) < f^{-1}(f(y)) \iff x < y$.

the path $A \to B \to D$, so it is not the case that a shortest path for weights w is also a shortest path for weights $f(w)$.

5 Problem 5

Algorithm. (Optimal Cutting Helper.) This algorithm takes as input some 3-tuple `size` $= (r, s, t)$ as well as two dictionaries, `prices` and `cache`, both of which map 3-tuples to integers.

1. If `size` is in `cache`: Return `cache`[`size`].
2. Create an empty array `smaller`.
3. For (r_i, s_i, t_i) in `prices.keys`:
   (a) If $r \geq r_i, s \geq s_i, t \geq t_i$: Append (r_i, s_i, t_i) to `smaller`.
4. If |`smaller`| $= 0$: Return (0, "TERMINATE").
5. Else:
   (a) Create a float `max_val` $= 0$.
   (b) Set `move` = "TERMINATE".
   (c) If `size` is in `prices`: Set `max_val` = `prices`[`size`].
   (d) For (r_i, s_i, t_i) in `smaller`:
      i. Set `nr` $= \log(r/r_i)/\log(2)$.
      ii. Set `ns` $= \log(s/s_i)/\log(2)$.
      iii. Set `nt` $= \log(t/t_i)/\log(2)$.
      iv. If any one of `nr`, `ns`, or `nt` is not an integer: Continue.
      v. Else if `nr` = `ns` = `nt` $= 0$: Continue.
      vi. Else:
         A. Set (`curr_cut`, `embedded`) = "Optimal Cutting Helper" called on (r_i, s_i, t_i) with the same `prices` and `cache`.
         B. Set `curr_cut` to itself multiplied by $2^{\texttt{nr}+\texttt{ns}+\texttt{nt}}$.
         C. If `curr_cut` > `max_val`: Set `move` = 'X'∗`nr`+'Y'∗`ns`+'Z'∗`nt` and `max_val` = `curr_cut`.
   (e) Set `cache`[`size`] = (`max_val`, `move`).
   (f) Return `cache`[`size`].

Algorithm. (Optimal Cutting Pattern.) We will use the above helper algorithm to solve the problem. We denote by `prices` the input dictionary mapping each dimension $r_i \times s_i \times t_i$ to its respective price u_i, and we denote by `size` the 3-tuple of dimensions (X, Y, Z) of the block.

1. Create an empty dictionary `cache`.

2. Call "Optimal Cutting Helper" on `size` with `prices` = `prices` and `cache` = `cache`. Return the second entry in the output 2-tuple.

Proof of correctness. The algorithm uses a bottom-up dynamic programming approach to solve the problem.

We are given a block of dimensions $r \times s \times t$ and a catalogue mapping dimensions to prices. Observe that if, for all dimensions (r_i, s_i, t_i) in the price catalogue, we have that $r < r_i, s < s_i$, or $t < t_i$, then our block cannot be cut into any marketable sizes, and so we must return that we can profit 0 from this block with the corresponding step simply being to "TERMINATE".

Assume now that $r \geq r_i, s \geq s_i$, and $t \geq t_i$ for at least one i. Then we can build a list `smaller` of the dimensions (r_i, s_i, t_i) from the price catalogue that are "feasible," or that have that $r \geq r_i, s \geq s_i$, and $t \geq t_i$. To determine whether some given dimensions (r_i, s_i, t_i) are actually obtainable from $r \times s \times t$ via a sequence of cuts, we require that

$$\frac{x}{2^k} = y \iff \frac{x}{y} = 2^k \iff \frac{\log(x) - \log(y)}{\log(2)} = k \in \mathbb{Z}, \tag{1}$$

where this equation must hold for each $(x, y) \in \{(r, r_i), (s, s_i), (t, t_i)\}$. If $k \notin \mathbb{Z}$, then we cannot halve the dimensions of the current block to arrive at $r_i \times s_i \times t_i$.

Once we determine that a given dimension is actually obtainable from the current block, we can then recursively call the algorithm "Optimal Cutting Helper" using the dimensions (r_i, s_i, t_i). Suppose this call returns some value $f(r_i, s_i, t_i)$ that is in fact the optimal profit we can obtain from a block of dimensions $r_i \times s_i \times t_i$, and along with that figure it returns the cutting steps followed to obtained that profit.

Then, for each set of dimensions (r_i, s_i, t_i) found to be obtainable from (r, s, t), we can find a corresponding value $f(r_i, s_i, t_i)$. Denote by k_r, k_s, k_t the values of k from (1) for $(x, y) = (r, r_i), (s, s_i), (t, t_i)$, respectively. Then it must be the case that the profit we can receive from cutting our block to (r_i, s_i, t_i) is given by $f(r_i, s_i, t_i) \times 2^{k_r \cdot k_s \cdot k_t}$. This is because k_r, k_s, k_t yield the number of cuts needed in each dimension to arrive at (r_i, s_i, t_i), so by reducing (r, s, t) to (r_i, s_i, t_i), we in fact get $2^{k_r \cdot k_s \cdot k_t}$ copies of (r_i, s_i, t_i).

Thus, we can write that the maximal profit we can obtain from a block of dimensions (r, s, t) will be given by the maximum of these recursively computed maximal profits times the appropriate number of blocks that will exist after cutting. The corresponding steps embedded steps can then be appended to 'X'$*k_r$+'Y'$*k_s$+'Z'$*k_t$, the steps followed to cut the block down to smaller blocks of size (r_i, s_i, t_i).

It only remains to be shown, then, that our base case holds. In the event that a block (r, s, t) has dimensions in the price catalogue, but the block cannot be cut down any further, then our algorithm simply returns `prices`$[(r, s, t)]$, which is trivially the maximal profit one can obtain from that block. Thus we

know from the argument above that this recursive algorithm outputs the correct maximal profit and corresponding cutting steps to arrive at that profit for any given block of dimensions (r, s, t).

Complexity. The algorithm caches the maximal profit obtainable from each block dimension (r_i, s_i, t_i) and only considers dimensions from `prices` that it can obtain from (r, s, t). Thus letting $|\texttt{prices}| = n$, we have that our helper algorithm will be called $\mathcal{O}(n)$ times, and in each call, it will loop over (at most) those n dimensions once again, yielding an overall time complexity of $\mathcal{O}(n^2)$.