CS 38, Midterm

Codename: Jambul

May 12, 2023

1 Problem 1

(a) Observe that a feasible set of roads is maximal $\iff$ the set of roads induces a triangulation of the circular room, which we visualize as a convex n-gon, where the work stations against the walls of the room are the figure's vertices. We assume $n \geq 2$, since for $n \leq 1$, there do not exist any feasible sets of roads. Let $G = (V, E)$ be the graph induced by the work stations (the vertices) and the roads (edges) in any given maximal, feasible set.

The equivalence relation stated above arises from the following observation: If a set of roads does not induce a triangulation of the n-gon, then there exists some convex polygon in G with at least four vertices. Connecting any two points in this convex polygon will not induce an intersection, so any maximal set of roads must be a triangulation of the n-gon. Conversely, if a feasible set of roads is not maximal, then it must be the case that we can add an edge to G without creating an intersection; this can only be done if there is a subgraph of G which is a convex polygon with at least four points.

We therefore seek the total number of ways to triangulate a convex n-gon. From elementary graph theory,[1] this is given by the $(n-2)$th Catalan number,

$$C_{n-2} = \binom{2(n-2)}{n-2} - \binom{2(n-2)}{n-1} = \frac{1}{n-1}\binom{2n-4}{n-2}, n \geq 2.$$

From Stirling's approximation,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \mathcal{O}(n^{-1})\right),$$

so

$$C_{n-2} = \frac{1}{n-1}\binom{2n-4}{n-2} = \frac{1}{n-1}\frac{\sqrt{2\pi(2n-4)}((2n-4)/e)^{2n-4}(1+\mathcal{O}(n^{-1}))}{[\sqrt{2\pi(n-2)}((n-2)/e)^{n-2}(1+\mathcal{O}(n^{-1}))]^2}$$

$$= \frac{1}{n-1}\frac{4^{n-2}}{\sqrt{n-2}\sqrt{\pi}}\frac{1}{1+\mathcal{O}(n^{-1})}.$$

[1] Ma 6b with Conlon is an awesome class.

In this last expression, it is easy to see that the 4^{n-2} term will dominate as $n \to \infty$. Thus the number of maximal, feasible sets of roads in the room will grow like 4^n, which is $e^{\Omega(n)}$, since $4 > e$.

(b) **Algorithm.** (Optimal Triangulation.) We will use dynamic programming to solve the problem. We require as inputs the number n of work stations in the room and a $n \times n$ matrix T with entries $T(i,j) = B_{i,j} > 0$.

```
1. Create an empty $n \times n$ matrix $o$.
2. Create an empty $n \times n$ matrix $e$.
3. Set $o(i,i) = 0 \ \forall \ i$.
4. For $i = n-1$ to 1:
    (a) For $j = i+1$ to $n$:
        i. If $j = i+1$: Set $o(i,j) = T(i,j), e(i,j) = [(i,i+1)]$.
        ii. If $j = i+2$: Set $o(i,j) = T(i,i+1)+T(i+1,j)+T(i,j), e(i,j) = [(i,i+1),(i+1,j),(i,j)]$.
        iii. Else:
            A. Set $o(i,j) = \max_{i+1<k<j+1}\{T(i,k)+o(i+1,k-1)+o(j,k+1)\}$.
            B. Taking $k^*$ to be the argmax of the above expression, set $e(i,j) = [(i,k^*)] + e(i+1,k^*-1) + e(j,k^*+1)]$.
5. Return $e(1,n)$.
```

Proof of correctness. As explained, the room is described by the graph $G = (V,E)$. The algorithm above makes use of the following observation: At any $v \in V$, the maximum feasible set that includes v will be the set that chooses the vertex w that v will connect to and then takes the maximum of the $B_{v,w}$ summed with the weights that can be obtained from the remaining triangles.

We also note that it will never be suboptimal to take edges between immediately adjacent stations, as these will never cause intersections in the graph, because the vertices form a convex arrangement (the room is circular). Thus we may start by taking the edge between vertices $n-1$ to n. Furthermore, the maximum weight of any 3 immediately adjacent points will always be the sum of the weights of the triangle formed between them.

We then iterate backward over $i = n-1, ..., 1$ and forward over $j = i+1, .., n$. We want $o(i,j), i < j$ to contain the maximum weight of a triangulation on the polygon $i, ..., j$. Thus we check each possible vertex $k, i+1 < k < j+1$ that i can connect to. At each of these connections, the maximum weight of the triangulation will be given by the maximum of $B_{i,k} + o(i+1,k-1) + o(k+1,j)$, as the first term accounts for the edge itself, $o(i+1,k-1)$ accounts for the weight of the triangulation clockwise of i, and $o(k+1,j)$ accounts for the weight of the counterclockwise triangulation.

In this way, we are certain that entry $o(i,j)$ will contain the maximum weight of a triangulation on a polygon on $i,...,j$. Since the prompt asks specifically for the edges that form this max-weight triangulation, we note that we create the matrix e solely to store the edges corresponding to the entries in o, so $e(i,j)$ contains the edges of the max-weight triangulation on vertices $i,...,j$.

Complexity. The two outer loops take $\mathcal{O}(n)$ time each, and finding the maximum inside the nested loop will take $\mathcal{O}(n)$ time. Thus the runtime of the algorithm is $\mathcal{O}(n^3)$.

2 Problem 2

Algorithm. (Constraint Satisfiability.) Let our inputs be $x_1,...,x_n$, Q, and N such that each entry $(i,j) \in Q$ corresponds to the constraint that $x_i = x_j$, and $(k,l) \in N$ to $x_k \neq x_l$. We may then use the following algorithm to solve the problem.

```
1. Create an empty $n \times n$ table $T$. We will use 0-indexing to index $T$.
2. Set all diagonal entries in $T$ to True.
3. For each pair $(i,j)$ in $Q$: Set $T(i-1,j-1) = T(j-1,i-1) =$ True.
4. For $i = 0$ to $n-1$:
    (a) For $j = 0$ to $n-1$:
        i. If $T(i,j) =$ True : Continue.
        ii. For $k = 0$ to $n-1$:
            A. If $T(i,k) = T(k,j) =$ True: Set $T(i,j) = T(j,i) =$ True.
5. For each pair $(i,j)$ in $N$: If $T(i-1,j-1) =$ True : return False.
6. Return True.
```

Proof of correctness. We will show that, following the main loop of the algorithm, the matrix T is populated such that $T(i,j) = \texttt{True} \iff x_i = x_j$. When we set $T(i,j)$ to be some value, we implicitly mean that both $T(i,j)$ and $T(j,i)$ are equal to that value, since the symmetry of T is maintanied in the algorithm.

Note that $(i,j),(j,k) \in Q \implies x_i = x_k$. More generally, we have that $(i_1,i_2),(i_2,i_3),...,(i_{m-1},i_m) \in Q \implies x_{i_1} = x_{i_m}$. Thus we may first populate the entries of the table such that $(i,j) \in Q \implies T(i,j) = \texttt{True}$.

Then we may iterate over the entries of the table once more; if $T(i,j) = \texttt{True}$ on this second iteration, we may pass over this cell, as we cannot set the values of T to anything other than `True` when using entries from Q. On the other hand, if

$T(i,j) = $ `None`, then it may be the case that in fact $x_i = x_j \implies T(i,j) = $ `True`. Since then $(i,j) \notin Q$, this could only be the case if there is some sequence $(i, i_1), (i_1, i_2), ..., (i_m, j) \in Q$.

We therefore may iterate over the columns of the row i we are currently in; if for any column $k, 0 \leq k < n$, we have that $T(i,k) = T(k,j) = $ `True`, then it must be the case that $T(i,j) = $ `True` as well. Note that we need only check two "steps," since, if $(i, i_1), ..., (i_{m-2}, i_{m-1}), (i_m, j) \in Q$, then it will be the case that $T(i, i_1) = T(i, i_2) = ... = T(i, i_{m-1}) = $ `True` and $T(i_m, j) = $ `True` by the time we arrive at entry (i,j).

Thus indeed $T(i,j) \iff x_i = x_j$ according to the rules in Q. Then we may simply check each rule in N against the respective entry in T; if the entry is marked `True`, we have a violation; otherwise, we do not.

Complexity. All three loops iterate over $\mathcal{O}(n)$ values, so the total runtime of the algorithm is $\mathcal{O}(n^3)$, which is indeed efficient. All other steps are dominated by these three nested loops.

3 Problem 3

Let $p(x) = 1 + x, q(x) = 4 + 3x^2$. We will represent these polynomials by the vectors $p = (1,1,0,0,0,0,0,0), q = (4,0,3,0,0,0,0,0)$, so $|p| = |q| = 8 = 2^3$. Furthermore, let $\omega = e^{i2\pi/8} = 1/\sqrt{2} - i/\sqrt{2}$. Then let

$$W = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^7 \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{14} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^7 & \omega^{14} & \dots & \omega^{49} \end{pmatrix}.$$

From lecture, we know that W is simply the Fourier transform over $\mathbb{Z}/8$. Thus to find the DFTs of p and q, we simply compute

$$\mathcal{F}(p) = P = p \cdot W, \mathcal{F}(q) = Q = q \cdot W$$

$$\implies P = (1+1, 1+\omega, 1+\omega^2, ..., 1+\omega^7),$$

$$Q = (4+3, 4+3\omega^2, 4+3\omega^4, ..., 4+3\omega^{14}),$$

where

$$\omega^k = \left(\frac{1}{\sqrt{2}}\right)^k (1-i)^k, k \in \mathbb{Z}$$

follows directly from the expression given for ω. We can then write R to be the pointwise product of P and Q, so

$$R_i = P_i \times Q_i = (1 + \omega^{i-1})(4 + 3\omega^{2(i-1)}), 1 \leq i \leq 8.$$

$$\implies R = ((1+1)(4+3), (1+\omega)(4+3\omega^2), ..., (1+\omega^7)(4+3\omega^{14})).$$

Finally, then, we know again from lecture that the inverse W^{-1} of W is given by

$$W^{-1} = \frac{1}{8}\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-7} \\ 1 & \omega^{-2} & \omega^{-4} & \dots & \omega^{-14} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-7} & \omega^{-14} & \dots & \omega^{-49} \end{pmatrix}.$$

Using this, we can then find the coefficients of the product of p and q by simply computing

$$R \cdot W^{-1} = (4, 4, 3, 3, 0, 0, 0, 0),$$

corresponding to the polynomial $4 + 4x + 3x^2 + 3x^3$, which indeed aligns with the expected result.

4 Problem 4

In the fractional knapsack problem, we wish to find a subset S of some set of items $\{1, ..., n\}$ alongside a list of corresponding fractional weights for the items in S such that $\sum_i S_i v_i$ is maximized, and $\sum_i S_i w_i \leq W$.

Consider the matroid $M = (U, \mathcal{F})$, where U contains the item numbers $\{1, ..., n\}$, and $\mathcal{F}$ contains all possible (unordered) subsets of U. Note that $\mathcal{F}$ then contains all sets of items from U for which there exist a feasible solution. For any set of items in U, assuming $W > 0$, we can always pick arbitrarily small fractions of the weights associated with the items in the set such that the sum of the weights times their fractions satisfies the capacity constraint. Thus

$$(i_1, ..., i_m) \in \mathcal{F} \iff c_1 w_{i_1} + ... + c_m w_{i_m} \leq W$$

for some constants $0 < c_1, ..., c_m \leq 1$. We will now verify that M is indeed a matroid.

We assume that we are given at least one item, so $\{1\} \in U \implies \{1\} \in \mathcal{F} \neq \emptyset$. Furthermore, if $J \in \mathcal{F}$ and $I \subseteq J$, then it must be the case that $I \subseteq U$, so $I \in \mathcal{F}$. Finally, suppose $I, J \in \mathcal{F}$ with $|I| > |J|$. Then I must have some elements that J does not; adding any of these elements to J will create another subset of U, which will thus be in $\mathcal{F}$, so M is indeed a matroid.

Now define the function $f(i) = v_i/w_i$. Then, we may simply sort the items in U in non-increasing order according to $f(i)$ and pop the first element x from U onto some list L as long as $L \cup \{x\} \in \mathcal{F}$ until the sum of the weights associated with the elements in L surpasses W; at this point we return L. Each time we add an element to L, we set its fractional weight to be as large as possible without surpassing W. In this way, at termination, all items in L but the last will have fractional weights of 1, while the last item will have the fractional weight that makes the sum of the fractionally-weighted items equal to W.

The greedy algorithm for this problem presented in lecture was shown to have a runtime of $\mathcal{O}(n \lg n)$, as the initial sorting step dominated all others.

Here, however, we must first generate $\mathcal{F}$. Doing this will take $\mathcal{O}(2^n)$ time, as there are 2^n possible subsets of the items in U. We then sort U according to $f(i)$ and apply the greedy property through a loop, which is a single pass over U. Thus the dominating step is the creation of $\mathcal{F}$, so the runtime of the algorithm is $\mathcal{O}(2^n)$. From this analysis, we can also see that this "generic" greedy approach is exactly the same as that presented in lecture except for the fact that it must generate $\mathcal{F}$, which is ultimately the limiting step of the algorithm.

5 Problem 5

Algorithm. (Local Minimum.) We will solve the problem using a divide-and-conquer approach. We assume we are given an $m \times n$ input matrix f.

1. Compute $i_r = \lfloor m/2 \rfloor, i_c = \lfloor n/2 \rfloor$.
2. Set $r = \min_j\{f(i_r, j)\}, c = \min_i\{f(i, i_c)\}$ so r and c are the minimum values of f across row i_r and column i_c, respectively.
3. Set $p = \min\{r, c\}$, and let (i_p, j_p) be the coordinates of entry p.
4. Set L to be the list of neighbors of p. (We do not consider entries that wrap around the input matrix, so corner entries have 2 neighbors, edge entries which are not corner entries have 3, and all other entries have 4.)
5. If p is less than or equal to all elements in L, return (i_p, j_p).
6. Set $m = \min\{L\}$. Denote (i_m, j_m) the coordinates of m.
7. If $i_m < i_r$ and $j_m > i_c$:
    (a) Recursively call "Local Minimum" on the upper right quadrant of the input matrix. Denote (n_i, n_j) the output of this recursive call.
    (b) Return $(n_i, n_j + i_c + 1)$.
8. If $i_m > i_r$ and $j_m < i_c$:
    (a) Recursively call "Local Minimum" on the bottom left quadrant of the input matrix. Denote (n_i, n_j) the output of this recursive call.
    (b) Return $(n_i + i_r + 1, n_j)$.
9. If $i_m > i_r$ and $j_m > i_c$:
    (a) Recursively call "Local Minimum" on the bottom right quadrant of the input matrix. Denote (n_i, n_j) the output of this recursive call.
    (b) Return $(n_i + i_r + 1, n_j + i_c + 1)$.
10. If $i_m < i_r$ and $j_m < i_c$:

(a) Recursively call "Local Minimum" on the upper left quadrant of the input matrix. Denote (n_i, n_j) the output of this recursive call.

(b) Return (n_i, n_j).

Proof of correctness. We will prove the correctness of the algorithm by showing first that it isolates a quadrant of the matrix with a local minimum, and second that it finds that local minimum.

The algorithm first finds the minimum elements in the middle row and the middle column of the matrix. It then takes the smaller of these two, which we denote p, as its starting point. If p is the smallest amongst its immediate neighbors, then clearly it is a local minimum, so we may return its coordinates. Otherwise, it must be the case that one of the neighbors of p is smaller than it, so we may then move to this element.

This element must reside in one of the four quadrants of the input matrix, since we started by scanning the middle column and middle row of the matrix, splitting the matrix into its four quadrants. Furthermore, since p cannot reside on the middle row or middle column (or else either p was not the smallest element across both the middle row and middle column or p was a local minimum), it must be the case that we are indeed now in one of those four quadrants.

We know that there will exist a local minimum in this quadrant. This is because either the new element m we have now moved to is smaller than all its neighbors, or there is a smaller surrounding it that we may move to. We will never leave this quadrant, since we know that m must be smaller than all elements in the middle row and middle column of the matrix, so since we can only move to progressively smaller elements, we are guaranteed to terminate at a local minimum of f. We may therefore recursively call this same algorithm on the new quadrant of f to continue the search for a local minimum.

Finally, then, we can check that we will indeed terminate at a local minimum by noting that, in the worst case, either we have recursed to the point that the input matrix is a single column or row vector. Finding the minimum of this column or row will trivially return the correct result. Otherwise, we would have returned a local minimum in an earlier step following the logic above. Thus the algorithm is guaranteed to terminate at a local minimum, assuming one exists.[2]

Complexity. In the first call, the algorithm will make $2n$ comparisons to find the local minima of the middle row and column. The algorithm will then recursively call itself on one of the quadrants of the matrix. This call will result in $\mathcal{O}(2n/2)$ comparisons, as the size of the quadrant will be a fourth (or less) of the original matrix. Summing over all calls, the algorithm will require

$$2n + \mathcal{O}(2n/2) + \mathcal{O}(2n/4) + ... + \mathcal{O}(1)$$

comparisons, which is indeed $\mathcal{O}(n)$.

[2]If this is not the case, then the algorithm will simply not return anything.