CS 170

midterm retrospective,

MSTs

1 Peachy algorithms

Rosa is a peach farmer. She has to tend to n fields, and all fields are ready for harvest! She must decide in which order to harvest the fields, as she can only harvest one field of peach trees a day. Due to differing soil and geographical conditions, each field has a different water requirement (that stays the same for all days) if it's not harvested, with field i needing $\ell_i > 0$ liters of water per day.

(a) Describe an algorithm that allows Rosa to harvest the fields in such an order that she minimizes the amount of water used by all fields together.

(b) Prove that your algorithm is correct.

(c) Analyze the runtime of your algorithm.

Solution:

(a) Sort the fields in decreasing order of water requirement using any standard sorting algorithm (from most water needed per day to least); harvest the fields one-by-one in this order.

(b) Suppose there is an optimal order of harvesting the fields which is not sorted this way; that is, there are some fields i and j where $\ell_i < \ell_j$ but i is harvested before j. Call the day on which field i is harvested d_i, and the day on which j is harvested d_j; then what we are assuming is $d_i < d_j$. The total amount of water used on both these fields is $\ell_i d_i + \ell_j d_j$. If the positions of these fields in the harvesting order were swapped, they would use $\ell_i d_j + \ell_j d_i$; the total amount of water saved by doing this swap is $\ell_i d_i + \ell_j d_j - \ell_i d_j - \ell_j d_i = (\ell_j - \ell_i)(d_j - d_i)$ (no other fields are affected since only these two change position in the order). Since $\ell_j - \ell_i > 0$ and $d_j - d_i > 0$, the total amount of water $(\ell_j - \ell_i)(d_j - d_i)$ saved by the swap must be positive, meaning the swap improved the quality of the ordering. This is a contradiction since we assumed that the unsorted order was optimal and so could not be improved upon.

(c) In order to output the correct order, our algorithm only needs to call a sorting algorithm; using, say, MergeSort, this takes $O(n \log n)$ time.

2 Gameshow

You are participating in a gameshow in which you are presented with a set of m boxes in a line, each containing an arbitrary unique number. Your goal is to find a box whose number is larger than that of the box to its left and the box to its right (unless it's box 1 or m, in which case it needs to be larger than just the number of the box it is next to), while opening as few boxes as possible.

(a) Describe an algorithm to find a box whose number is bigger than that of both of its neighbors. Your solution should run faster than $O(m)$.

(b) Prove that your algorithm is correct.

(c) Analyze how many boxes your algorithm opens in the worst case. Your final answer should be of the form $O(f(m))$, where f is some function that you specify.

Solution:

(a) If the current $m \leq 3$ then simply open all boxes and return the box whose number satisfy the conditions. Otherwise we open box $\lfloor \frac{m}{2} \rfloor$, $\lfloor \frac{m}{2} \rfloor + 1$, $\lfloor \frac{m}{2} \rfloor + 2$ (the *middle three* boxes on the line). Let their numbers be a, b, c respectively. If $b > a$ and $b > c$ then we return the middle box. Otherwise if $b < a$ then perform the algorithm recursively on boxes number $1, \dots, \lfloor \frac{m}{2} \rfloor$; if $b > a$ and $b < c$ perform the algorithm recursively on boxes number $\lfloor \frac{m}{2} \rfloor + 2, \dots, m$.

Intuitive explanation: This problem is commonly known as the 1D peak-finding problem. We always "climb the hill" or recurse on the half with the larger number when trying to find a peak. A complete proof would mention that recursing on this half guarantees a peak value in either of the following cases: (1) the numbers in that half are in increasing/decreasing order, which means that the last element is guaranteed to be a peak element or (2) the numbers increase until some point i, which implies that $i - 1$ is the peak element. If we go downhill, we might not find a peak (there could be a lake at the bottom of the hill). Another fact that could be used along with the previous statements when writing the proof is that any list has a max element, which is also a peak element.

(Note: The notion of "middle three" boxes is not unique. Any viable definition for "middle three" boxes is fine. If $b < a$ and $b < c$ then it doesn't matter which half to recurse, as long as we recurse it on *only* one half).

(b) We proceed via strong induction on m.

Base Case: If $m \leq 3$, then the algorithm is trivially correct. We can always find our goal in constant time since we opened all boxes.

Induction Step: If the middle box satisfy our conditions, then the algorithm is correct since we returned the middle box. If $b < a$, then we can view the left half of the line as its own individual line of boxes since we already know the left box (box $\lfloor \frac{m}{2} \rfloor$) has a bigger number than the middle box, so we simply need the left box to be larger than the other box it is next to. Therefore by the inductive hypothesis the algorithm returns the correct box in the left half of line. A similar argument goes for the case where $b < c$ and we recurse on the right half.

(c) The base case is a constant time procedure. Every recursive layer we perform a constant time procedure and one recursive call with the parameter halved. Therefore we have

$$T(m) = T\left(\frac{m}{2}\right) + O(1)$$

Therefore the overall time complexity is $O(\log m)$.

3 Greatest Roads Revisited

Quentin wants to travel from San Francisco to New York by bus. Assume that there are $n-2$ cities he could change buses at on his way (so n cities total), with a total of m direct bus rides between pairs of cities (assume each ride is one-way). The ith bus ride has price p_i and all prices are positive integers. Quentin wants to travel as cheaply as possible, while booking a trip such that the sum of all bus ride prices is a multiple of 5 dollars, as he hates small change.

Find an algorithm that finds the cheapest way to get from San Francisco to New York while paying a multiple of 5 dollars for the trip.

(a) Describe your algorithm.

Hint: Think about how you can use a similar approach to what you used in a homework problem to solve this question. Furthermore, since all prices are integers, note that there are really only 5 cases you need to consider. It might also help to first consider a case where Quentin wants to end up with an even price.

(b) Prove that your algorithm is correct.

(You can use any algorithm from lecture as a sub-routine, and you do not have to re-argue its correctness.)

(c) Analyze the runtime of your algorithm.

Solution:

(a) We model the problem as a graph $G=(V,E)$, where the vertices are the cities and the edges are the bus routes, each having a weight p_e. We now create a graph G': We create 5 copies of the vertex set V and label them 0 through 4. Now, for edge (i,j), we connect vertex i in copy k to vertex j in copy $k+p_{(i,j)} \bmod 5$ with an edge of cost $p_{(i,j)}$ for each k. Intuitively, this means that the copy corresponds to the current small change modulo 5. Then, we run Dijkstra's algorithm on the resulting graph to find a path from node s in copy 0 to node t in copy 0.

(b) We note that by the construction of the graph, if we end at node t in layer 0, the sum of all prices is equal to 0 modulo 5, i.e. a multiple of 5. Hence, suppose there is a shorter path p in G that has this property. Hence, we derive a path p' in G': We begin at s in copy 0. Whenever we take an edge in G, we take the equivalent edge in G', moving to the appropriate copy. Since the path returns a cost that is a multiple of 5, we end at t in copy 0. Hence, we have constructed a valid path p' in G' that is shorter than our original graph, which is a contradiction.

(c) Note that we execute Dijkstra's algorithm on a graph with $5n$ vertices and $5m$ edges. Hence, the runtime is $O(5(m+n)\log 5n) = O((m+n)\log n)$. Meanwhile, creating the copies of the graph is linear in the number of edges, so the time to create G' is dominated by the runtime of Dijkstra. Hence, the runtime is $O((m+n)\log n)$.

4 Striped matrices

A *striped matrix* is an n-by-n matrix A such that $a_{ij} = a_{i-1,j-1}$ for $i = 2, 3, \ldots, n$ and $j = 2, 3, \ldots, n$. For example this is a striped matrix:

$$\begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 1 & 3 & 5 \\ 4 & 2 & 1 & 3 \\ 6 & 4 & 2 & 1 \end{bmatrix}$$

(a) Show how to represent a striped matrix using $O(n)$ space instead of the $O(n^2)$ representation shown above.

(b) Describe an $O(n \log n)$-time algorithm for multiplying an n-by-n striped matrix A (represented in your $O(n)$-space format from Part (a)) by a vector $\vec{v}$ of length n.

(c) Assume you have a working solution to Part (b). Using that solution, describe an $O(n^2 \log n)$ algorithm for multiplying an n-by-n striped matrix A with an n-by-n matrix, M.

Extra credit (1 pt): Describe a faster algorithm if M is also a striped matrix. Do not try this unless you have extra time.

Solution:

(a) Given a striped matrix (also known as a Toeplitz matrix in literature) A, all the information of A is contained in the first row and first column (every other matrix element is determined by these elements and the relation $a_{ij} = a_{i-1,j-1}$). We can store the first column and first row of A in the following order:

$$\begin{bmatrix} a_{n,1} & a_{(n-1),1} & \cdots & a_{2,1} & a_{1,1} & a_{1,2} & \cdots & a_{1,(n-1)} & a_{1,n} \end{bmatrix}.$$

We store $2n - 1$ elements, using $O(n)$ space.

(b) To compute the matrix-vector product $A\vec{v}$, we first compute $a(x) \cdot v(x)$, where

$$a(x) = a_{n,1}x^{2n-1} + a_{(n-1),1}x^{2n-2} + \cdots + a_{2,1}x^{n+1} + a_{1,1}x^n + a_{1,2}x^{n-1} + \cdots + a_{1,(n-1)}x^2 + a_{1,n}x$$

and

$$v(x) = v_n x^n + v_{n-1}x^{n-1} + \cdots + v_2 x^2 + v_1 x.$$

Then $(A\vec{v})_i$, the ith element of $A\vec{v}$, can be read off as the coefficient of x^{n+i} in $a(x) \cdot v(x)$. Since the degrees of $a(x)$ and $v(x)$ are both $O(n)$, the polynomial multiplication $a(x) \cdot v(x)$ takes $O(n \log n)$ time using FFT.

(c) Use our algorithm from part (b) to compute the products of A with each column of M:

$$AM = A \begin{bmatrix} | & | & & | \\ \vec{m}_1 & \vec{m}_2 & \cdots & \vec{m}_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | \\ A\vec{m}_1 & A\vec{m}_2 & \cdots & A\vec{m}_n \\ | & | & & | \end{bmatrix}.$$

We compute $O(n)$ matrix-vector products, each of which take $O(n \log n)$ time, so our total runtime is $O(n^2 \log n)$.

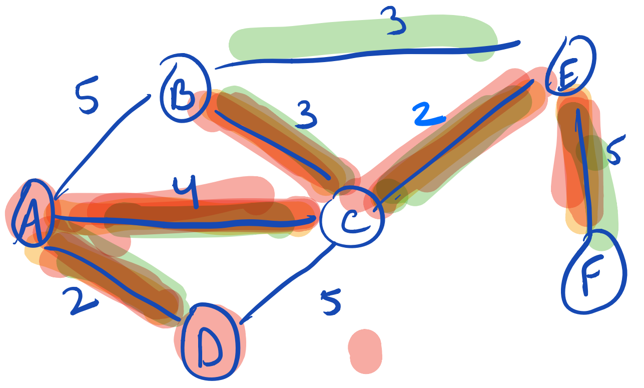
Extra Credit: While AM is not a striped matrix, we can compute each "stripe" diagonal of AM in $O(n)$ time. We do this by computing the upperleft-most entry of the stripe in $O(n)$ time, and computing each subsequent element of the diagonal recursively in $O(1)$ time. For example, the elements on the longest "stripe" diagonal (that is, elements of the form $(AM)_{i,i}$) can be computed in $O(n)$ time as follows:

$$\begin{aligned} (AM)_{1,1} &= a_{1,1}m_{1,1} + a_{1,2}m_{2,1} + \cdots + a_{1,n}m_{n,1} & &[O(n) \text{ work}] \\ (AM)_{2,2} &= (AM)_{1,1} - a_{1,n}m_{n,1} + a_{2,1}m_{1,2} & &[O(1) \text{ work}] \\ &\vdots & &\vdots \\ (AM)_{n,n} &= (AM)_{(n-1),(n-1)} - a_{1,2}m_{2,1} + a_{n,1}m_{n,2} & &[O(1) \text{ work}]. \end{aligned}$$

Since AM has $2n-1$ diagonals, we compute AM in $(2n-1) \cdot O(n) = O(n^2)$ time.

MSTs

- Minimum spanning tree — tree of edges in the graph that connects / spans all vertices.

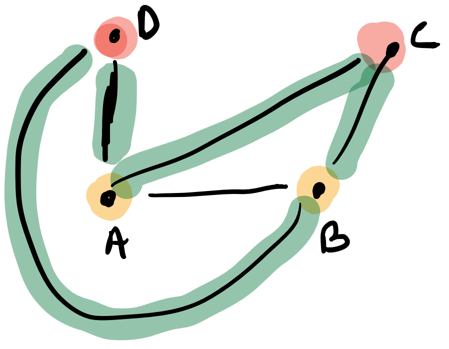


- |V|, |V| - 1 edges
- Trees are acyclic
- one path from a to b for any vertices a, b in graph
- removing any edge disconnects the tree.
- adding any edge will introduce a cycle.

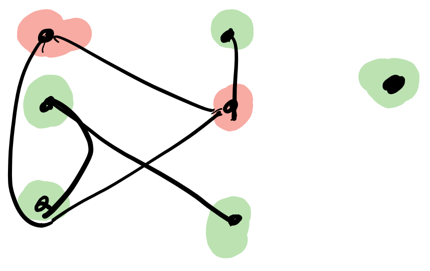
Cut property

- Cut : Assignment of vertices to one of two sets

- Crossing edge : edge that connects node from one set to node of the other.



With the cut {D, C}, {A, B}, the edges AD, AC, BC, DB cross the cut.



Cut property: given any cut, the minimum weight crossing edge is in the MST.

Proof?
- other MST that doesn't include smallest edge in cut.
- add the min. weight edge to the new MST
  - creates a cycle
  - Know that 2 edges cross the cut - go and come back to original edge.
- remove non-min weight edge from cycle to get new MST.
- contradiction!
  - now have new ST with weight less than the initial MST.

2 algorithms :

Prim's :

1. Start with an arbitrary node.
2. Add the shortest edge that already has one vertex in the MST.
3. Repeat until there are V-1 edges!

> Same implementation as Dijkstra's, except instead of the priority queue considering distance from the start vertex, it considers distance from the MST so far.

Runtime : $O(|V| + |E| \log |V|)$.

$E \sim O(V^2)$

Kruskal's

1. Sort all edges from lightest to heaviest
2. Take one edge at a time (in sorted order), and add to MST if it doesn't create a cycle.
3. Repeat until there are $V-1$ edges.

Runtime $O(\underline{|E| \log |E|})$ (sorting). $O(1)$ $2^{2^{2^{2^{2}}}}$ $\uparrow$

If pre-sorted, then $O(|E| \log^* |V|)$ $2 \uparrow 5$

essentially $O(\underline{|E|})$!

(see union - find data structure)

Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 MST Basics

For each of the following statements, either prove or give a counterexample. Always assume $G = (V, E)$ is undirected and connected. Do not assume the edge weights are distinct unless specifically stated.

(a) Let e be any edge of minimum weight in G. Then e must be part of some MST.

(b) If e is part of some MST of G, then it must be a lightest edge across some cut of G.

(c) If G has a cycle with a unique lightest edge e, then e must be part of every MST.

(d) For any $r > 0$, define an r-path to be a path whose edges all have weight less than r. If G contains an r-path from s to t, then every MST of G must also contain an r-path from s to t.

2 Updating a MST

You are given a graph $G = (V, E)$ with positive edge weights, and a minimum spanning tree $T = (V, E')$ with respect to these weights; you may assume G and T are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $\hat{w}(e)$. You wish to quickly update the minimum spanning tree T to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each, give a description of an algorithm for updating T, a proof of correctness, and a runtime analysis for the algorithm. Note that for some of the cases these may be quite brief.

(a) $e \notin E'$ and $\hat{w}(e) > w(e)$

(b) $e \notin E'$ and $\hat{w}(e) < w(e)$

(c) $e \in E'$ and $\hat{w}(e) < w(e)$

(d) $e \in E'$ and $\hat{w}(e) > w(e)$

3 A Divide and Conquer Algorithm for MST

Is the following algorithm correct? If so, prove it. Otherwise, give a counterexample and **explain why it doesn't work**.

```
procedure FindMST($G$: graph on $n$ vertices)
    If $n = 1$ return the empty set
    $T_1 \leftarrow$ FindMST($G_1$: subgraph of $G$ induced on vertices $\{1, \ldots, n/2\}$)
    $T_2 \leftarrow$ FindMST($G_2$: subgraph of $G$ induced on vertices $\{n/2 + 1, \ldots, n\}$)
    $e \leftarrow$ cheapest edge across the cut $\{1, \ldots, \frac{n}{2}\}$ and $\{\frac{n}{2} + 1, \ldots, n\}$.
    return $T_1 \cup T_2 \cup \{e\}$.
```

4 Huffman Proofs

(a) Prove that in the Huffman coding scheme, if some symbol occurs with frequency more than $\frac{2}{5}$, then there is guaranteed to be a codeword of length 1. Also prove that if all symbols occur with frequency less than $\frac{1}{3}$, then there is guaranteed to be no codeword of length 1.

(b) Suppose that our alphabet consists of n symbols. What is the longest possible encoding of a single symbol under the Huffman code? What set of frequencies yields such an encoding?