CS 170 DIS 10

Released on 2019-04-01

1 MST Mishap

You found the MST of a huge graph $G = (V, E)$, but afterwards you realized that you made a small mistake: the weight of a particular edge e is not $w(e)$, but instead $w'(e)$. Give a linear-time algorithm for finding the true MST. For each of the four cases, clearly describe your algorithm (with pseudocode if desired). Your algorithm must run in linear time.

(a) Claim : Let T be the original tree. There exists an MST T' such that T and T' differ on at most 2 edges.

(b) Case 1: e is in the current MST and $w'(e) < w(e)$.

(c) Case 2: e is in the current MST and $w'(e) > w(e)$.

(d) Case 3: e is not in the current MST and $w'(e) > w(e)$.

(e) Case 4: e is not in the current MST and $w'(e) < w(e)$.

Solution:

(a) Since only one edge e changed weight, the only possible change can be to remove e and add another edge e' such that T after the swap is still a tree. Suppose that we can add an edge e_1 and remove e_2 for some $e_2 \neq e$ such that the graph after the swap T' is a tree and $\text{cost}(T') < \text{cost}(T)$ in the new graph. Then $\text{cost}(T') < \text{cost}(T)$ in the original graph (since only e changed weight), contradicting that T was an MST.
Therefore, we only need to determine if removing e from T and adding another edge can produce a tree of smaller cost.

(b) **Algorithm:** Do nothing.
Proof: We decreased the cost of an edge $e \in T$. We know that originally, $\text{cost}(T) \leq \text{cost}(T')\ \forall T'$. Let newCost be the cost of the tree after reducing the weight of e, and let $d = w(e) - w'(e) > 0$. Then $\text{newCost}(T) = \text{cost}(T) - d \leq \text{cost}(T') - d \leq \text{newCost}(T')$ for all T', so T is still an MST.

(c) **Algorithm:** Remove e from T to make T', and run DFS on T' to determine the 2 connected components A and B of T' (removing one edge in a tree creates 2 connected components). Find the edge e' of minimal weight from A to B in the original graph, and add it to T' to get the true MST.
Proof: If we remove e from T, then we will split T into two connected components, and adding the edge e' of minimal weight between these two connected components will turn T into an MST by the cut property.

(d) **Algorithm:** Do nothing
Proof: We increased the cost of an edge $e \notin T$. We know that originally, $\text{cost}(T) \leq \text{cost}(T')\ \forall T'$. Increasing the cost of e can only increase the cost of each T', so $\text{cost}(T) \leq \text{cost}(T')\ \forall T'$ still holds. Thus, T is still an MST.

(e) **Algorithm:** Let $e = (u, v)$. We can run BFS on T starting from u to determine the unique path from u to v, and in particular the edge e' of maximum weight along this path. If $w(e') > w'(e)$, then remove e' from T and add e to T. Otherwise do nothing.
Proof: We only need to determine if adding e to T and removing another edge produces a tree of smaller cost. Since T needs to be a tree, we must remove an edge along the unique cycle created when adding e to T. To minimize the cost of T, this edge should be any edge of maximum weight (which could be e) in the path.

2 Scheduling Meetings : Greedy and dynamic

You are the scheduling officer for the CS department. The department has one conference room, and every day you receive a list L of n requests for it, in no particular order: $L = (s_1, e_1), \ldots, (s_n, e_n)$. Each item in the list (s_i, e_i) is a pair of integers representing start and end times, respectively, with $s_i < e_i$. Your goal is to schedule as many meetings as possible in one day, with the constraint that meetings cannot conflict with each other (the intervals for the scheduled meetings must be disjoint). Assume for simplicity that all of these integers are distinct.

(a) You consider a greedy algorithm for this problem: *Choose the meeting that starts first, and delete all conflicting meetings; repeat.* Give a simple example with just three meetings showing that this cannot work.

(b) Now find an optimal greedy algorithm: given the list L as specified above, your algorithm should return a list of non-conflicting meetings of maximum length.

(c) Briefly justify the correctness of your algorithm.

(d) What is the running time of this algorithm, given the unsorted list of n meetings? Justify briefly.

(e) Now suppose each meeting also has a positive integer weight w_i, and you want to schedule the set of meetings that maximize the total weight. Your input is the list L of n requests, again in no particular order, with each element now formatted as (s_i, e_i, w_i). Since the greedy algorithm no longer works, you design a dynamic programming algorithm to return the maximum weight achievable. (*Note*, you do not need to return the set of meetings that achieve it).

First, define your subproblems in words:

For $i = 1, \ldots, n$ $W[i] =$_____

(f) Give clear pseudocode for this problem. You do not need to justify your solution. What is the runtime of your algorithm?

Solution:

(a) (1, 10), (2,3), (4,5)

(b) While the list of meetings is not empty,

Pick the meeting with the smallest end time and add it to scheduled meetings.

Delete all meetings with start times before this end time.

Return the list of scheduled meetings

(c) Compare the greedy solution G with some other solution S. We will show that G is at least the size of S, and thus that G is no worse than any solution, and thus is an optimal solution.

Consider G and S in increasing order of meeting end times g and s, and consider the location i of the first difference between them. Either $g_i > s_i$ or $g_i < s_i$. The former is impossible: G and S are equal before the ith term, and g_i is the smallest end time of the remaining set. Regarding the latter, we could simply change s_i to g_i, swapping out the ith meeting that S had for the one G had. This choice fits the first i ending earlier than they were otherwise, which allows the other meetings in S to still be valid but perhaps allow more meetings.

In this manner, we could convert S to G while keeping it no worse than before, so G is at least as good as any solution.

(d) $n \log n$, easily achieved by sorting the list and doing a linear sweep, keeping track of the most recent end time added.

(e) There's a trick here that the list is not yet sorted. Cleanest answer is:

For $i = 1, \ldots, n$ $W[i]$ the maximum total weight achievable by scheduling tasks that end no later than the ith task sorted by end time. We'll also accept the same answer without reference to sorting, as long as their pseudocode sorts it. Also there's an equivalent answer that goes in reverse order and uses start time.

(f) (a) Sort the list L by end times.

(b) Set $W[0] = 0$

(c) For i from 1 through n, use binary search to find index j, which is the highest index such that $e_j < s_i$, or 0 if none exists. Then set $W[i]$ to the maximum of $(w_i + W[j])$ and $W[i-1]$

(d) Return $W[n]$.

It takes $\Theta(n \log n)$ time to sort the list by n time, and $\log n$ time for the binary search for each index, so that the third step also takes $n \log n$ time. Overall algorithm takes $\Theta(n \log n)$ time. Note, there's a tweak to efficiently (in linear time) find all the j values, but it doesn't affect asymptotic runtime.

3 Midterm Prep: Short Answer

Answer each question below *concisely*

(a) Suppose that in a weighted graph with negative weights we know that the shortest path from s to t has at most $\log |V|$ edges. Can we find this shortest path in $O(|V|^2 \log |V|)$ time?

(b) True/False : In a Huffman code, if there is a leaf of depth 1 and a leaf of depth 3, then there must also be a leaf of depth 2.

(c) True/False : In a Huffman code where all letter frequencies are distinct, the third least frequent letter has depth either equal to that of the two least frequent letters, or one less.

(d) If a linear program is unbounded, then its dual must be unbounded.

(e) If the capacities of a max flow problem are multiples of $\frac{1}{2}$, the value of the optimum flow will also be a multiple of $\frac{1}{2}$.

(f) If we add an integer $k > 0$ to all capacities of a max-flow problem, the max flow is also increased by exactly k.

(g) If we multiply all capacities of a max flow problem by an integer $k > 0$, the max flow is also multiplied by exactly k.

Solution:

(a) Yes; simply use Bellman-Ford for $\log |V|$ iterations; it is guaranteed to find all paths at least of length $\log |V|$, and since $E = O(|V|^2)$, achieves the desired runtime.

A common mistake was to just take the usual Bellman-Ford running time of $O(|V||E|)$ and replace the $|E|$ with $\log |V|$. This is wrong because the $|E|$ is the number of edges in the graph, not just the number of edges in the path.

(b) False, consider the frequencies 0.5, 0.125, 0.125, 0.125, 0.125. This results in a leaf of depth 1 (0.5) and four of depth 3, and none of depth 2.

(c) True. If it had a depth less than two less than that of the two least frequent letters, then we could swap it with some other letter at a greater depth, which must exist as a descendant of the sibling of the location where the two least frequent letters merge in the tree, which would result in a more efficient code, contradiction. And if it had a depth greater than the two least frequent letters, we could swap it with one of them.

(d) False, consider the LP: max x such than $-x \leq 1$ and x is nonnegative. The dual is to minimize y such that $-y \geq 1$ and y is nonnegative. The primal is unbounded, and the dual is infeasible.

(e) True, the amount Ford Fulkerson increases the flow by at each stage is a multiple of 1/2.

(f) False, consider S to A and S to B with capacity 0, and A to T and B to T with capacity 0. We increase max flow by 2k when we add k to all capacities.

(g) True, the Ford Fulkerson algorithm would find exactly the same flow, which would now be k times as large.

4 Edge Disjoint Paths

Given a directed graph $G = (V, E)$ and two vertices in it a and b, find out the maximum number of edge disjoint paths from a to b. Two paths are said to be edge disjoint if they don't share any edge.

Solution: Take the graph G and assign a capacity of 1 to each edge. Run Ford-Fulkerson on this graph with a as the source vertex and b as the sink vertex. The max flow is equal to the max number of edge disjoint paths. This is correct because every edge gets used in at most one path and the bottleneck capacity of any path is 1 (since all edges have capacity one) and so we augment one unit of flow along every path and don't consider any edge along that path again.