CS 180 HW 3

Narek Daduryan

November 23, 2021

1

Given an undirected weighted graph G with n nodes and m edges, and we have used Prim's algorithm to construct a MST T. Suppose the weight of one of the tree edges $(u, v) \in T$ is changes from w to w', design an algorithm to verify whether T is still a MST. All weights are distinct.

Solution. Notice that there are two cases here: if $w' < w$, then we know that T is still the MST. This is because the MST represents the path through all vertices of G such that the sum of the weights is minimized. Let W be the sum of the weights of T in the original graph. If we then update the weight of one tree-edge and reduce it by c (s.t. $w' = w - c$), then the total weight of T will by $W - c$ which must still be the minimum of all paths, since the total weight was reduced.

Now consider the case where $w' > w$ (the weight is increased). Intuitively, in this case the MST might change, since the total weight W of the path has increased, there may be another path whose total weight is now the minimum.

In this case, there are another two subcases: if T is still the MST, then the edge (u, v) remains in T. Otherwise, the edge (u, v) is removed from T, and replaced by another edge. To find this edge, notice that removing (u, v) from T will create two disjoint subsets (two partitions of T that are disconnected). There is the partition that contains vertex u and the partition that contains the vertex v. In order to create a spanning-tree, we need to choose one edge that connects these two partitions; and in order to create a new minimum-spanning tree, we need to select the minimum-weight edge that goes between these two partitions.

Here is an example algorithm that can construct the two partitions of T after removing edge (u, v), and then find the minimum edge e' between the two partitions. (Note, if $|e'| = w'$, then we know the original T is still a valid MST)

Note, in this algorithm, T is the current MST, G is the graph with vertices V and edges E, L is the set of weights (reflecting the updated weight w', and u and v represent the updated edge (u, v).

```
1  function isStillMST(T, G=(V,E), L, u, v):
2      removing edge (u,v) from T
3      part_u = BFS on T from u
4      part_v = BFS on T from v
5
6      newWeight = w'
7
8      for each vertex x in part_u:
9          for each edge (x,z) of x:
10             if z in part_v:
11                 newWeight = min(newWeight, L[(x,z)])
12
13     return newWeight == w'
```

Note: let's assume we can use a data structure such as a hash set when constructing "part_u" and "part_v" so that we can have constant time lookup on line 10.

Now let's analyze the time complexity of the algorithm. Line 2 is a constant-time operation, or can be $O(m)$ depending on how T is stored, if we have to linearly search for the specific edge to remove it. Lines 3 and 4 will do a BFS algorithm on T, and store each vertex visited in a hash set or another structure. The BFS itself will take $O(n + m)$ time. Then we have a for-loop over each vertex in part of T, and for each of those vertices, we iterate through their edges. Thus, there will be the $\sum_{v \in V} degree(v) = |E| = m$ iterations of this loop body. The loop body on lines 10 and 11 performs a lookup for a vertex (which can be constant-time if "part_v" is a hash set), and then updates the newWeight by looking up the weight of the edge; this is also constant time. Thus, the entire loop body is constant time, and thus the entire loop runs in $O(m)$ time.

Since we are given that the given graph G has a spanning tree, then we know that $m \geq n-1$. (Since we need at least enough edges to span all vertices). Thus, we can use this fact to simplify our time complexity. We know that the entire algorithm will run in $O(n + m)$ (BFS: $(n + m)$ and for-loop: $O(m)$). However, since we have the constraint that $m \geq n - 1$, we can state that the entire time complexity is $O(m + m) = O(m)$.

□

2

Let $G = (V, E)$ be an undirected graph and each edge $e \in E$ is associated with a positive weight l_e. Assume weights are distinct. Prove or disprove:

2 (a)

Let P be a shortest path between two nodes s, t. Now, suppose we replace each edge weight l_e with (l_e^2), then P is still a shortest path between s and t.

Solution. False. Prove by contradiction. Let $G = (V, E)$, with $V = \{s, q, t\}$. Assume there are two paths from s to t: $P_1 = (s, q, t)$ and $P_2 = (s, t)$. The weights of the edges are as follows: $l_{sq} = 2$, $l_{qt} = 3$, $l_{st} = 4$. Thus, the total weight of $P_1 = l_{sq} + l_{qt} = 2 + 3 = 5$. The total weight of $P_2 = l_{st} = 4$. Thus, the shortest path P from s to t is P_2 with a total weight of 4.

Now, let G' be the same as G, where each edge's weight is squared ($l'_e = (l_e)^2$). Now example the same paths P_1 and P_2 from s to t. The weight of $P_1 = l_{sq}^2 + l_{qt}^2 = 4 + 9 = 13$. The weight of $P_2 = l_{st}^2 = 16$. Thus, in graph G', the shortest path is P_1, not P_2. Thus, we have shown by contradiction that when squaring all the edge weights, a shortest path P in the original graph is not still a shortest path in the new graph.

In general, for two paths of length n and m, we can see that $\sum_{i=0}^{n} a_i < \sum_{j=0}^{m} b_i$ does not imply that $\sum_{i=0}^{n} a_i^2 < \sum_{j=0}^{m} b_i^2$. (Corollary 1) For $n = 1$ and $m = 2$, we have that $a + b < c$ does not imply that $a^2 + b^2 < c^2$. □

2 (b)

Let T be a minimum spanning tree for the graph with the original weight. Suppose we replace each edge weight l_e with (l_e^2), then T is still a MST.

Solution. True. Given the conclusion of the part (a), we know that squaring individual elements does not maintain the length relation between two arbitrary paths (Corollary 1). However, the MST is not an arbitrary path. If we analyze the MST through an algorithm such as Prim's algorithm, we can see that squaring each individual weight does not modify the algorithm's execution. Let's take a look at Prim's algorithm, which simply chooses an edge e_i at each iteration to construct the MST at the end of the algorithm. If at every step of the algorithm, Prim's algorithm always chooses the same e_i, then at the end, the MST is the same. Thus, let's see if there is any different in running Prim's algorithm on $G = (V, E, L)$ and $G' = (V, E, L')$ where L' represents the weights, each the square of the original weights L.

Let's look at any arbitrary iteration of Prim's algorith, where we are choosing an edge e_i to add to our MST. At this arbitrary iteration, we have our minimum-spanning-tree-so-far, T, and we have a set of edges $\bar{E}$ between T and a node not in T. Per Prim's algorithm, we choose e_i to be the smallest weight edge within $\bar{E}$. Now, notice that if we have a sequence of weights (in $\bar{E}$) $l_0, l_1, ..., l_n$, and the sequence of square-weights $l_0^2, l_1^2, ..., l_n^2$, if we sort both sequences, we obtain

the same sorting for both sequences. In other words, if we have $a < b < c...$, then we also have that $a^2 < b^2 < c^2....$ Thus, for Prim's algorithm, we choose the smallest weight, which for example would be the edge with weight a in G, or a^2 in G'. In our example, thus, for both G and G', at an arbitrary iteration of Prim's algorithm, we choose the same edge e_i to append to the MST. Thus, if we run Prim's algorithm on both G and G', we obtain the same MST. This proves that if P is an MST in G, and we square the weights, obtaining graph G', then P is still an MST. $\square$

3

Given a list of intervals $[s_i, f_i]$ for $i = 1, ..., n$, please design a divide-and-conquer algorithm that returns the length of the largest overlap among all pairs of intervals. The algorithm should run in $O(nlogn)$ time.

Solution. Intuitively, we can sort the intevals by their starting time, and then divide the given intervals in half by this starting time. Then, we can recursively find the max-overlap within these sub-problems. The only issue is, then we would not be searching for overlaps that exist between one interval in the left half and one interval in the right half. What we can do to find any such overlaps is consider only the right-most f_i that starts in the left-half of the array, and compare it with each of the elements that start in the right half. This is because we know any element in the left-half will not overlap on the left-side of the array, since all elements in the right-half of the array have s_i in the right half. Thus, by maximizing f_i, we are maximizing how "far" we reach into the right half of the array. Then, intuitively, this algorithm will run in $O(nlogn)$ by similar principle to merge sort, where we recursively call the algorithm twice on each half of the array, then have a merge step with takes linear time. Let's define this algorithm more concretely:

```
1
2  sort A by s_i
3
4  function MostOverlap(A):
5      let n = length of A;
6      if (n is 0 or 1) return 0;
7
8      A_left = left half of A;
9      A_right = right half of A;
10     left_most_olap = MostOverlap(A_left)
11     right_most_olap = MostOverlap(A_right)
12
13     // compute most overlap between interval in left half and right half
14     between_most_olap = 0;
15     l = element in A_left with largest finishing time
16     for each element r in A_right
17         between_most_olap = max(overlap(l, r), between_most_olap);
18
19     return max(left_most_olap, right_most_olap, between_most_olap);
```

Assume that getting the left/right half of an array involves dividing the length of the original array in two, and then using the correct floor/round operations when needed to correctly handle both odd and even cases.

Also, assume that "overlap" is a function that returns the overlap between two given overlaps. Note, this function can run in constant time, since finding

the overlap between two elements only consists of some constant-time arithmetic operations or conditionals.

Analyzing the time complexity of this, we know that MostOverlap() will recursively call itself twice, with each call having size $n/2$ (half the original size of n). Thus, let's analyze the merge step of this algorithm. To merge, we first find the largest finishing-time element in "A_left" which involves linearly searching through the array "A_left" and maintaining the current maximum element, and performing a comparison for each element. Thus, this operation is linear time to the size of "A_left", thus runs in $O(n/2) = O(n)$. Then, we have a for-loop which runs over each element in "A_right". In each iteration, we call the "overlap" function, which is constant-time, and we call the "max" function, which is constant time. Thus, the entire for loop runs in $O(n/2) = O(n)$ time. The last "max" call is also constant time. Thus, the entire merge step of the divide and conquer algorithm runs in $O(n)$ time. Thus, we have that $T(n) = 2T(n/2) + n$; and by master theorem (similar to merge sort), we know this is $O(nlogn)$.

□

4

Among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(nlogn)$ invocations of the equivalence tester.

Solution. We can solve this using a divide-and-conquer algorithm, similar in structure to a merge sort.

Given an array of n cards, we split it in two, and run our algorithm on both halves; this should return the element which comprises the majority of its half (occurs more than $m/2$ times, where m is the size of that half). If there is no majority element, it returns such a signal (for example, like null). Then given the majority elements of the left half and right half, we then do a linear search for these two elements, counting occurrences. If either element is found more than $n/2$ times, then that means that element is the majority, and we return it. Otherwise, then that means there does not exist a majority.

To prove this correctness, we need to show that if an array of n elements has a majority, then at least one half of that array must have that same element as a majority. The majority element occurs at least $n/2 + 1$ times. If we divide the array into two halves, of $m = n/2$ elements, then we know one of the halves has as least $m/2 + 1$ occurrences. For example, we can divide the majority into $m/2$ and $m/2 + 1$ halves, where one half contains exactly $m/2 + 1$ occurrences. Any other distribution of this element will result in a half containing more than $m/2 + 1$ occurrences. Thus, if the original array contains a majority, then we know at least one of its halves must contain that same element as a majority.

Thus, we define the following algorithm:

```
1  function findMajority(A):
2      let n = length of A
3      if (n is 0) return null;
4      if (n is 1) return A[1];
5
6      A_left = left half of A
7      A_right = right half of A
8      left_Maj = findMajority(A_left);
9      right_Maj = findMajority(A_right);
10     left_Maj_count = countOccurrences(left_Maj, A);
11     right_Maj_count = countOccurrences(right_Maj, A);
12
13     if (left_Maj_count > n/2) return left_Maj
14     if (right_Maj_count > n/2) return right_Maj
15
16     return null;
17
18
```

```
19 function countOccurrences(element, A):
20     count = 0;
21     for each element a in A:
22         if equivalenceTester(element, a) count += 1;
23     return count;
```

In the algorithm, lets assume returning an element of the input array returns a reference to that card, and equivalenceTester works on these references. Further, equivalenceTester(null, x) returns false for any x, and equivalenceTester(x, x) returns true for any x. Also, assume that getting the left/right half of an array involves dividing the length of the original array in two, and then using the correct floor/round operations when needed to correctly handle both odd and even cases.

Let's analyze the time complexity of this. Intuitively, it follows the same division-and-conquer pattern as a merge sort. We know that the function findMajority will divide its input into two each step; thus findMajority itself will run $log(n)$ times. The merge step within findMajority consists of two calls to countOccurrences; each call runs in $O(n)$, since this function consists only of a linear search through its input array. Thus, the merge step in findMajority takes $2O(n) = O(n)$ time. Thus, as we have seen in class, this divide and conquery algorithm that divides by two at each step, and takes $O(n)$ for its merge step, will run in a total of $O(nlogn)$ time

□