Homework 4

CS 4104 (Spring)

Assigned on March 3, 2021.
Submit PDF solutions on Canvas
by 11:59pm on March 10, 2021.

My team-mate is: Joseph McAlister

Problem 1 (30 points) Let $G = (V, E)$ be an undirected connected graph and let $c : E \to \mathbb{R}^+$ be a function specifying the costs of the edges, i.e., every edge has a positive cost. Assume that no two edges have the same cost. Given a set $S \subset V$, where S contains at least one element and is not equal to V, let e_S denote the edge in E defined by applying the cut property to S, i.e.,

$$e_S = \arg\min_{e \in cut(S)} c_e.$$

In this definition, the function "arg min" is just like "min" but returns the argument (in this case the edge) that achieves the minimum. Let F be set of all such edges, i.e., $F = \{e_S, S \subset V, S \neq \emptyset\}$. In the definition of F, S ranges over *all* subsets of V other than the empty set and V itself. Answer the following questions, providing proofs for all but the first question.

(i) (5 points) How many distinct cuts does G have? We will use the same definition as in class: a cut is a set of edges whose removal disconnects G into two or more non-empty connected components. Two cuts are distinct if they do not contain exactly the same set of edges. For this question, just provide an upper bound.

Solution: In general, the upper bound of the number of distinct cuts that a graph G with $n = |V|$ can have is given by $2^n - 2$.

(ii) (8 points) Consider the graph induced by the set of edges in F, i.e., the graph $G' = (V, F)$. Is G' connected?

Solution: Yes. Since F includes *all* possible subsets of $S \subset V$, the induced graph would be a layering of all the cheapest edges in the original set of edges E which, as given, *connect* G.

Proof: Assume for the sake of contradiction that the induced graph $G' = (V, F)$ is not connected.

Then, there exists some node $v \in V$ which is not connected any other node $v' \in V$.

However, by definition, F is the set of all edges selected by the edge cut property, ranging over *all* subsets of $S \subset V$.

Therefore, F necessarily includes the least costly edge in the chosen subset $S = \{v\}$ where e_S is the cheapest edge connecting v to any other vertex in $S - V$.

If G was connected to begin with, then F will contain the cheapest edge connecting $S = \{v\}$ to $S - V$.

However, this statement contradicts our assumption that G' is not connected, therefore G' is connected. ■

(iii) (7 points) Does G' contain a cycle?

Solution: No. Because of the edge cut property by which F is populated which will always select the cheapest edge connecting S to $S - V$. In order for G' to contain a loop, e_S would have to be incorrectly chosen based on the nodes in S.

Proof: Assume for the sake of contradiction that G' could contain a cycle.

Then, for some $S \subset V, S \neq \emptyset$ there must exist some node $v \in V - S$ for which the cost of the cheapest edge that would connect it to an edge in our cut(S) is some positive, unique value c_e.

This edge must be the cheapest connecting edge in order to be added to the connected component of G' that F describes.

Assume that by adding this edge to F, then G' will gain a cycle.

However, this edge will never be added, since there must exist some other edge connecting $v \in V - S$ to S of cheaper or equal cost, since by adding v we gain a cycle.

However, this is a contradiction since we have already asserted that adding the edge with cost c_e connecting v to S would form a cycle.

Therefore G' cannot contain a cycle. ■

(iv) (5 points) How many edges does F contain?

Solution: $|V|-1$ edges. Since S is strictly subset of V, therefore, constructively, it is not possible to contain more edges in F than there are nodes in V.

Proof: Assume for the sake of contradiction that G' is connected and has no cycles, but has $|F| \neq |V| - 1$.

Case 1: $|F| < |V| - 1$

We have shown above that G' is connected, and if there are fewer than $|V| - 1$ edges in G' then G would not be connected. (This was shown in Homework 2.1). This is a contradiction since we assumed that G' is connected.

Case 2: $|F| > |V| - 1$

We have also shown that G' does not contain any cycles, and if there are more than $|V| - 1$ edges, then there would be a cycle on G'. (This was also shown in Homework 2.1). This is a contradiction since we assumed that G' does not contain any cycles.

(v) (5 points) What conclusion can you draw from your answers to the previous statements?

Solution: G' is an MST since it uses the minimum amount of the cheapest edges to fully connect G.

Problem 2 (35 points) You return home for the weekend all agog with the exciting new ideas you have discovered in the algorithms class. You tell your evil twin about the Minimum Spanning Tree (MST) problem and the clever algorithms for computing it. Your sibling pooh poohs your new-found wisdom and proposes the following simple algorithm on to compute the MST of an undirected graph G, assuming that no two edges have the same cost.

1. Maintain a set T of edges. Initially T is empty.
2. Process the edges of E in *any* order.
3. For each edge $e \in E$,
   (a) Add e to T.
   (b) If T contains a cycle, delete e from T.

Something seems fishy. Could this algorithm really compute the MST? Show up your sibling by fixing the algorithm so that T is indeed the MST at the end and prove that the modified algorithm computes the MST of G. What you should include in your solution is both the corrected algorithm and its proof of correctness.

Notes: (a) The algorithm is not the same as Kruskal's algorithm since it processes the edges in *any* order. In contrast Kruskal's algorithm processes the edges in increasing order of cost. (b) In your fix, you decide not to sort the edges by cost or use any data structure such as the priority queue to sort the edges by cost. (c) We are interested only in proving the correctness of this algorithm. We are not interested in its running time. *Most of the points are for a clear and complete proof of correctness.*

Hint: I am providing an elaborate hint here. Your proof of correctness should show that at the end of the algorithm, T is an MST. Therefore, you have to prove all three points implied by the phrase "Minimum Spanning Tree." Prove each of these statements:

Solution: The modification I would make to my nefarious twin's algorithm would be to add a for loop to step 3 part (a) as follows:

1. Maintain a set T of edges. Initially T is empty.

2. Process the edges of E in *any* order.
3. For each edge $e \in E$,
   (a) Add e to T.
   (b) If T contains a cycle, then delete the edge with the highest cost in the cycle accordingly:

```
   Input: cycle the list of edges in a cycle
   /* e is the edge we just added to T                                   */
1  max_edge ←e
   /* find the edge with the highest cost in the cycle                   */
2  foreach edge e_i ∈ cycle do
3  |  if c(e_i) > c(max_edge) then
4  |  |  max_edge ←e_i
5  |  end
6  end
7  Remove max_edge from T
```

(a) T **does not contain a cycle.** This proof should be easy.

Solution: By construction, it is not possible for T to contain a cycle since, on line 7 of the adjusted algorithm, one edge in a cycle is deleted whenever a cycle is created, thus breaking said cycle.

(b) T **is spanning, i.e., connects all vertices in** G**.** This part can be challenging. Consider an arbitrary subset S of V. It is enough to prove that at the end of the algorithm, T contains at least one edge in cut(S).[1] As the modified algorithm progresses, what can you show about that edges in cut(S) that are also in T? Informally, I am suggesting that you imagine the algorithm is running in the background while you focus your attention on the edges in cut(S). The algorithm will process these edges in some order. Think about this order to show that at the end of the algorithm, T contains at least one edge in cut(S).

Solution: For any subset $S \subset V$, at least one edge in cut(S) will be contained in T. As the adjusted algorithm processes all nodes in G, it is guaranteed that at least one of the nodes in T will be connected by an edge in cut(S).

Proof: Assume for the sake of contradiction that T does not contain any edges in cut(S).

The presented algorithm process all edges in E and adds them (and presumably their corresponding nodes) to T provided that they do not create a cycle. If a cycle is created, the edge of maximal cost in the cycle will be deleted from T.

We know that at some point an edge $e \in$ cut(S) will processed.

From this point onward, e can only be removed from T if a cheaper edge is found that also connects T to the node that e connected.

In either case (e does or does not get replaced by a cheaper edge), e or its replacement will be connected to T by an edge in cut(S).

However this contradicts our assumption that T will not contain any edges in in cut(S), therefore T will contain at least on edge in cut(S).

(c) T **is an MST.** If you have proven the first two parts, then you know that T is a spanning tree. How many edges can it contain? If you modified the algorithm correctly, then what can you say about the edges that you did not include in T? Remember that the algorithm has processed every edge in G. Now combine what know so far with what you proved in part (v) of Problem 1.

Solution: T is an MST.

We have shown that T is *minimum* insofar as it contains no cycles (using the minimum number of edges to connect the graph G), and all connections are of minimal cost, as, since all edges

[1] For every subset S, if T contains at least one edge in cut(S), then T is connected. You may assume this fact.

are processed, any cheaper connection which forms a cycle will result in the more expensive edge being removed from T.

We know that T is *spanning* since our algorithm will try to add all edges to T, guaranteeing that the node will be connected (only removing edges that create cycles, but none that simply connect G).

We know that T is a *tree* since it has $|V| - 1$ edges (per problem 1, and T having no cycles, but connecting all the nodes of G).

Therefore T is an MST.

Problem 3 (35 points) You are given two sets of n points. The first set of points $\{p_1, p_2, \ldots, p_n\}$ lies on the line $y = 0$. The other set of points $\{q_1, q_2, \ldots, q_n\}$ lies on the line $y = 1$. None of these point sets is sorted by x-coordinate. Now construct a set of n line segments as follows: for each $1 \leq i \leq n$, connect p_i to q_i. Develop a divide-and-conquer algorithm that computes how many pairs of these line segments intersect. Your algorithm should run in $O(n \log n)$ time.

Solution: Let $P = [p_1, p_2, ..., p_n]$, and $Q = [q_1, q_2, ..., q_n]$.

We begin by sorting P according to the x-coordinate in $O(n \log n)$ time and arranging the points in Q such that all of the indices match that of their corresponding points in P.

We can then treat inversions in Q is intersections between pairs of line segments. That is, for indices $i < j$, an inversion/intersection occurs when $Q[i] > Q[j]$.

Then, we can apply a Divide and Conquer strategy using modified versions of the `merge_and_count` and `sort_and_count` algorithms to find the number of intersections in $O(n \log n)$ time.

First, we apply the `sort_and_count` algorithm on recursive halves of Q to sort and count inversions in partitions.

Then we count inversions that occur across partitions using the `merge_and_count` algorithm.

Algorithm 1: `merge_and_count`

Input: A, B: two sorted lists
Output: L: the merged list of A, B and r the number of inversions/intersections

```
r ← 0
L ← [|A| + |B|]
pointer_a, pointer_b ← 0, 0
while A ≠ ∅ and B ≠ ∅ do
    a_i, b_i ← A[pointer_a], B[pointer_b]
    if a_i < b_i then
        Add a_i to L
        pointer_a ++
    else
        Add b_i to L
        pointer_b ++
        r+ = (|A|− pointer_a)
    end
end
Add remainder of non-empty list to L
return r, L
```

Algorithm 2: `sort_and_count`

Input: L: A list of x-coordinates for the points in Q
Output: The sorted list S and the number of inversions in L

$n \leftarrow |L|$
if $n \leq 1$ **then**
  return 0
end
$A \leftarrow L[0, \lceil n/2 \rceil]$
$B \leftarrow L[\lceil n/2 \rceil + 1, n]$
$(r_A, A) =$ `sort_and_count`(A)
$(r_B, B) =$ `sort_and_count`(B)
$(r, L) =$ `merge_and_count`(A, B)
return $r = r_A + r_B + r, L$

Proof of Correctness: Every inversion in Q (and corresponding intersection in line segments formed by connecting points in Q to P) is counted exactly once, and no non-inversions are counted.

Base case: $n = 1$, there are no inversions.

Inductive hypothesis: the presented algorithms count the number of inversion correctly for all set of $n - 1$ or fewer line segments

Inductive step: Consider an arbitrary inversion of points $i, j \in Q$ such that $i < j$, but $x_i < x_j$. This inversion is counted in precisely three mutually exclusive locations:

1. $i, j \leq \lfloor n/2 \rfloor : x_i, x_j \in A$ is accounted for by r_A by the inductive hypothesis
2. $i, j \geq \lceil n/2 \rceil : x_i, x_j \in B$ is accounted for by r_B by the inductive hypothesis
3. $i \leq \lfloor n/2 \rfloor, j \lceil n/2 \rceil : x_i \in A, x_j \in B$ is accounted for when x_j is output by `merge_and_count`

Furthermore, no non-inversion are counted as when any $x_j \in B$ is added to the merged list, it is smaller than all remaining elements in A, since A is sorted.

Proof of runtime complexity:

The running time of the presented algorithm is $O(n \log n)$ as the modifications to the standard $O(n \log n)$ merge sort algorithm which add the `_count` functionality are all constant time operations.