

# Homework 4

ECE345 - Group 16

November 24th, 2023

Total pages: 9

Team Member	Student Number
Ardavan Alaei Fard	1007934620
Rowan Honeywell	1007972945
Isaac Muscat	1007897135

## Contents

<b>Question 1</b>	<b>2</b>
(a) . . . . .	2
(b) . . . . .	2
<b>Question 2</b>	<b>3</b>
<b>Question 3</b>	<b>4</b>
<b>Question 4</b>	<b>5</b>
<b>Question 5</b>	<b>6</b>
(a) . . . . .	6
(b) . . . . .	7

## Question 1

(a)

To show that  $G$  and  $H$  share the same set of minimum spanning trees, we can't show that for any sub-MST, they both share the same safe edges. In theorem 23.1 in CLRS 3rd edition, it is shown that the light edge crossing the cut of any cut of  $G$  that respects  $A$  is safe for  $A$ . The theorem uses the fact that since  $(u, v)$  is a light edge,  $w(u, v) \leq w(x, y)$ . This means  $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$ . If both  $w(x, y)$  and  $w(u, v)$  get incremented by one when translating  $G$  to  $H$ , the above theorem shows that  $(u, v)$  is still the safe edge for  $G$  and  $H$ . Therefore,  $G$  and  $H$  share the same minimum spanning trees.

(b)

$G$  and  $H$  do not share the same shortest paths between all pairs of vertices. As a counter example, consider a graph  $G$  with a destination vertex  $v_d$  and source vertex  $v_s$ . We need only to show that the shortest path from  $v_s$  to  $v_d$  is not the same for both graphs. Let there be two paths connecting the two vertices. In graph  $G$ , one path consists of 8 edges with weights of 1 while the other path consists of one edge with a weight of 9. In graph  $G$ , the shortest path involves traversing the path with 8 edges for a cost of 8 instead of traversing the single edge with a cost of 9. In contrast in graph  $H$ , the 8 edge path has each edge weighted at 2 with a path cost of 16 and the single edge is weighted at 10. Therefore, the shortest path will be achieved by traversing the single edge. Therefore, the shortest path from  $v_s$  to  $v_d$  is different in  $G$  and  $H$ .

## Question 2

The shortest path from every node in North America  $v_{NA}$  to every node in Europe  $v_E$ , must include the edge  $e_{transatlantic}$ . Additionally, both  $v_{NA1}$  and  $v_{E1}$  will be in the set of vertices for each shortest path since it is the only path connecting the two subgraphs  $G_{NA}$  and  $G_E$ . Using the above statements, the shortest path from each  $v_{NA} \in G_{NA}$  to each  $v_E \in G_E$  can be broken down into the following:  $shortestpathfromv_{NA}tov_{NA1} + shortestpathfromv_{E1}tov_E + e_{transatlantic}$ .

The shortest path from  $v_{NA}tov_{NA1}$  can be found in  $\mathcal{O}((V+E) \lg V)$  using Dijkstra's algorithm (CLRS 3rd Ed, pg 661 - 662) with  $e_{transatlantic}$  removed. Similarly, the shortest path from  $v_{E1}tov_E$  can be found in  $\mathcal{O}((V+E) \lg V)$  using Dijkstra's algorithm with  $e_{transatlantic}$  removed. The shortest path can then be found by concatenating the previous two paths found with  $e_{transatlantic}$ .

Since two  $\mathcal{O}((V+E) \lg V)$  operations are run at most (in fact the amortized cost is less - not asymptotically - because both subgraphs only run Dijkstra's algorithm on their nodes and edges which is less than  $E$  and  $V$ ), the asymptotic time complexity of the algorithm is  $\mathcal{O}((V+E) \lg V)$ . Dijkstra's algorithm uses a priority-queue of vertices to determine the next vertex with the smallest cost which takes  $\mathcal{O}(V)$  space.

## Question 3

First, take each conversion rate and construct a directed graph using an adjacency list. Each node represents a currency and its edges represents the conversion ratio from the currency of node A to node B. For instance, inserting the conversion CAD to USD involves creating two edges with each edge specifying the conversion from and to both currencies. An adjacency list is used because it keeps BFS and DFS down to a  $\mathcal{O}(V + E)$  time complexity, it takes asymptotically less space and time to create than an adjacency matrix. To resolve a currency exchange query, a DFS or BFS (both have take  $\mathcal{O}(V + E)$  time) is run on the first node in the exchange query and terminating if the second node is found in the search. If DFS/BFS fail to find the second currency, then there is no conversion possible. If the search is successful, the currency exchange rate will be the muliplication of each edge weight. The algorithm takes  $\mathcal{O}(V + E)$  time because for adjacency lists, inserting an edge and a vertex takes constant time and  $V + E$  insertions are done to construct the graph. The time complexity of the search is also  $\mathcal{O}(V + E)$ . The algorithm takes  $\mathcal{O}(V + E)$  space because only an adjacency list is used to store the currency exchange rates and their relationships.

## Question 4

## Question 5

(a)

To preface, we will characterize a “bad” swap as the swapping of a heavy child to a right child position. For each left-heavy node (more descendants on its left than its right) that we come across in the traversal of the right-most branch, it is guaranteed that a bad swap will occur as it will be swapped to a right child position.

Next, because we only ever traverse rightwards when merging, every time we come across a left-heavy node, the **maximum** possible number of nodes that remain to be traversed is divided by two. Since this is the maximal case, there are **at most**  $\lg n$  left-heavy nodes in the right-most branch and consequently a maximum of  $\lg n - 1$  bad swaps that can occur (since we do not swap the last node).

**Charge & Allocation:** Let us charge  $2 \lg n - 1$  credits for each merge operation, where  $n$  is the total number of nodes in both  $h$  and  $h'$ . We will **deposit** one credit for each of the first  $\lg n$  nodes that are merged along the right-most branch. Additionally, we will **deposit** one credit for each bad swap that is performed. Finally, we will **charge** one credit for each swap (good or bad).

**Credit Invariant:** For each bad swap and corresponding heavy child in a right child location, there will be an extra credit stored. Also, one credit will be stored for each of the first  $\lg n$  nodes that must be merged and subsequently swapped.

**Argument for Positive Credit:** A perfect **leftist** heap will have a right-most branch of maximum size  $\lg n$ . Since we deposited one credit for each of the first  $\lg n$  merged nodes, the cost of swapping each of these nodes will be covered. Of course, there may be additional

nodes in the right-most branch since skewed heaps swap unconditionally. Each bad swap creates a longer right-sided branch somewhere in the heap which may be traversed in a future merge operation. It is important to note that the next time this branch is traversed, it will engage a **good** swap, as the right-heavy node will be swapped to the left. Since one credit was deposited for each bad swap, and there are a maximum of  $\lg n - 1$  bad swaps that can occur (as shown previously), these self-correcting swaps will be covered by what was previously deposited.

Therefore, all of the swapping operations can be accounted for by an initial deposit of  $\lg n + \lg n - 1 = 2\lg n - 1$  credits. It has thus been shown that SKEWEDHEAPMERGE has an amortized time-complexity of  $\mathcal{O}(\lg n)$ .

## (b)

**Analysis of INSERT:** In the case of inserting a new node into a skewed heap, we can simply employ the SKEWEDHEAPMERGE algorithm from part (a). This works because inserting a single node into a skewed heap equivalent to merging two heaps, the only condition being that one is only a root node.

**Analysis of DELETEMIN:** Similar to the analysis of INSERT, we can also use SKEWEDHEAPMERGE to delete the minimum element from a skewed heap. This can be done by disconnecting the heap's root node (minimum) from its two children and merging the resulting left and right sub-heaps.

In conclusion, since both INSERT and DELETEMIN can be performed through a single merge, they both have the same amortized time-complexity of  $\mathcal{O}(\lg n)$ .