Assignment1

Student number: 490051481

September 22, 2024

Problem 1

Before answer this question, I would like to briefly describe about the Chat-GPT answer. Below, I will use "he" to represent Chat GPT.

AI Output:

Step 1 is ambiguous, he did not mention:

- how to solve collision problem
- how to implement hash map
- how to set (key, value) pairs for hash map

My recommended improvement is below:

- because all integers of array V are distinct, therefore, we can directly use *price* of products as *map_v*'s key without considering collision problem.
- I will use AVL Tree to implement this map. From lecture slide, we know get and put operation take $O(h)$ time, which can be $O(logn)$ if the tree is balanced (i.e. AVL tree). In addition, each node of AVL tree stores the pair we described below. Lastly, AVL tree will be sorted according to *price* and the hash function is identity function (i.e. $f(x) = x$).
- using *price* as *map_v*'s *key* and *index* as *map_v*'s *value* where *price* is the integer in array V and *index* is its correspond index in array. Namely, the node structure for AVL is $(price, index)$. By the way, although we did not use *index* here, I still choose to keep it because I do not want to break the rule for Map.

After initialized the hash map *map_v*, we can search *price* in *map_v* with at most $O(logn)$ time complexity.

Step 2 is correct, it just calculate all possible summation between elements in X and elements in Y.

Step 3 is run in the inner for-loop. As for Binary search, it only works for sorted list, but we do not have sorted list here, therefor, I think we need not to use binary search. The only thing we need to do is judging whether "*sum*"s exist in *map_v* for all *sum*. If none of them inside *map_v*, then return $false$. Otherwise, return $true$.

Step 4 does not make sense to me, I suggest delete this step. I think we already make sure you and your friend contribute exactly one coin in Step 2

Step 5 can be simplified. As I mentioned before, my recommended Step 3 already check whether there is a such product with proper price.

Correctness Proof:

All part does not make sense, I know nothing from his description. He should clearly specify each step's purpose and why we can judge whether there is a required product. My suggestion is "In step 1, I use AVL tree to implement the hash map *map_v* which store (price, index) pair in each node where price and index comes from array V. In addition,

AVL tree will be sorted according to *price*. In step 2, I use two nested loop to get all possible summation *sum* between coin in *X* and coin in *Y* and try to find it in *map_v*. If successfully match one or some *sum* to *map_v.price*, that means there is an item in the vending machine whose price is exactly equal to the sum of two coins, one from you and one from your friend. As a result of that, *true* will be returned. False will be returned otherwise".

Time Complexity analyze:

Hash Map construction time complexity is incorrect, because we need to insert n node into *map_v* and insertion operation for AVL tree takes $O(logn)$, therefore, it takes $O(nlogn)$.

Nested Loops time complexity is correct.

I think we do not need to use binary search.

It needs include time complexity for search operation in *map_v* (i.e. AVL tree) which takes $log(n)$.

The overall complexity $n^2log(n)$ is correct but description should be " dominated by the nested loops and the AVL search operations"

(a)

I think AI response does not fully answer the question. Although his answer exactly runs in required time complexity, some of its strategies are not necessary and does not make sense. For example, it need not to include binary search method.

(b)

I think there are many parts that are ambiguous. For example, he did not mention about how to implement hash map. By the way, I point them out in the above briefly description part.

(c)

The correctness argument is totally incorrect. Reader can not know why his algorithm is correct after read it. I also include a suggested version in the above briefly description part.

(d)

Except for the Hash Map construction (the time complexity for hash map initialization is vary), analysis for nested loop and binary search time complexity is correct. In addition, the overall time complexity for the provided algorithm is correct if his hash map is AVL-tree based Hash map. Lastly, I also give suggestion in the above briefly description part.

Problem 2

(a)

I will use one max-heap and one AVL tree to solve this question. The structure of node in AVL is $(KEY : ID, VALUE : address\ for\ correspond\ router\ in\ max\text{-}heap)$. The structure of node in max-heap is $(KEY : avg\_delay,\ Value : [ID,\ total\_delay,\ number\_of\_data])$.

The AVL tree is similar to the Week 4 lecture slide one. So it still has $Put(KEY)$, $Delete(KEY)$, $Find(KEY)$ operation methods with $O(log(n))$ time complexity. Notice that those operations include rotation step which is used to obey the property of AVL, for detail, please see lecture 4. I will use ID to "arrange" AVL. The mainly purpose of this AVL is to implement the $CURRENT - DELAY(ID)$ method. And we will not use it to implement $MAX - DELAY()$.

The max-heap is similar to the week 5 lecture slide one. According to post #565 in ED, it has $Insertion()$ and $Removing()$ operations. Notice that, these two methods include a sequence of operations, such as $Upheap, Downheap, Search$. The purpose is to keep the property of max-heap, for detail, please see lecture 5. I will use avg_delay to "arrange" the max heap. The mainly purpose of this max-heap is to implement the $MAX - DELAY()$ method.

Now, let me introduce how it works for the required operations.

- $INITIALIZE()$: initializing an empty AVL tree and an empty binary tree will take exactly $O(1) + O(1) = O(1)$ time complexity.
- $ADD - DATA(ID, delay)$: There are two cases for this method.
  - Case 1: If we do not find any node in AVL by using $Find(ID)$ method.
    - Case 1.1: The $delay$ exceed X; In this scenario, we can simply do nothing because when we only have one data and the $delay$ higher than X means the average delay is higher than X as well. As a result, the total time complexity is $O(1)$.
    - Case 1.2: The delay for the router does not exceed X; In this scenario, we need to update both AVL and Max-heap. We firstly create node MAX_N, which I will use it in max-heap later, and set $MAX\_N.avg\_delay = delay$; $MAX\_N.value[0] = ID$; $MAX\_N.value[1] = delay$; $MAX\_N.value[2] = 1$. Secondly, We create node AVL_N, which I will use it in AVL later, and set $AVL\_N.ID = ID$; $AVL\_N.value = address\ of\ MAX\_N$. Next, I will use $Put(ID)$ to put the node AVL_N into AVL which takes $O(logn)$. Lastly, I will add MAX_N to max-heap by using $Insertion()$ which takes $O(logk)$. The time complexity for initializing 2 nodes is $O(1)$. In addition, since there is no loop, the time complexity is $O(logk) + O(logk) = O(logk)$ for put 2 nodes in each tree respectively. In total, we have $O(1) + O(logk) = O(logk)$.
  - Case 2: If we successfully use $Find(ID)$ method find the existing node (i.e. router) in the AVL. For convenience, we call it AVL_N and its correspond router in max-heap is MAX_N

* Case 1.1: The average delay for the router exceed X: In this case, we need to delete router in both AVL and Max-heap. Before doing that, we need to update the value in max-heap. Firstly, We can find AVL_N by using $Find(ID)$ in AVL and use $AVL\_N.value$ to find the MAX_N. Secondly, we add $delay$ to $MAX\_N.value[1]$, add 1 to $MAX\_N.value[2]$, and set $MAX\_N.key = \frac{MAX\_N.value[1]}{MAX\_N.value[2]}$. Since in this case, $MAX\_N.key > X$, therefore, we choose to use $Removing()$ (including related steps to keep the property of max heap) method to remove this router from max-heap. Lastly, we use $Delete(ID)$ to remove AVL_N. The time complexity for finding AVL_N is $O(logk)$ because tree has height k. The time complexity for finding MAX_N is $O(1)$ because directly use address. Node's value/key assignment steps take $O(1)$. The time complexity for deleting MAX_N takes $O(logk)$. The time complexity for deleting AVL_N is $O(logk)$. Since there is no loop, we can directly add them to get overall time complexity $O(logk) + O(1) + O(1) + O(logk) + O(logk) = O(logk)$.
* Case 1.2: The average delay for the router does not exceed X. Firstly, We can find AVL_N by using $Find(ID)$ and use $AVL\_N.value$ to find the MAX_N. Secondly, we add $delay$ to $MAX\_N.value[1]$, add 1 to $MAX\_N.value[2]$, and set $MAX\_N.key = \frac{MAX\_N.value[1]}{MAX\_N.value[2]}$. In this case, we need to do both $Upheap$ and $Downheap$ operation because $MAX\_N.key$ can become larger or become smaller. We need not to update AVL because node's key does not change and its corresponding node still exist in max-heap. The time complexity for finding AVL_N is $O(logk)$ because tree has height k. The time complexity for finding MAX_N is $O(1)$ because directly use address. Node's value/key assignment steps take $O(1)$. The time complexity for $Upheap$ is $O(logk)$. The time complexity for $Downheap$ is $O(logk)$. Since there is no loop, we can directly add them to get overall time complexity $O(logk) + O(1) + O(1) + O(logk) + O(logk) = O(logk)$.

- $CURRENT - DELAY(ID)$: we can just use $Find(ID)$ to find the correspond router with ID in AVL. Then just using $AVL\_N.value.key$ to get the average delay. The time complexity for finding the router AVL_N is $O(logn)$. The time complexity for finding the average delay is $O(1)$ because we just use address to directly get it. In total, the time complexity is $O(logn) + O(1) = O(logn)$.
- $MAX - DELAY()$: We can just return the root of max-heap. It takes $O(1)$.

(b)

My data structure is like a dictionary, where AVL tree store the "key" and max-heap store the "value", namely, you can directly access node in max-heap without traverse all nodes in it. In addition, whenever we add a new router to the network, we need to update both tree. We can also use AVL tree to find existing router and update its "values" which store inside the max-heap. If the average delay exceed X after updating, we can delete nodes from both trees. For more detail, please see part (a).

(c)

I already analyze the time complexity for each operation in part (a). For space complexity. Since there are only two trees and each tree has exactly k nodes which means the complexity is $O(k) + O(k) = O(k)$.

Problem 3

(a)

In my algorithm, each vertex has 2 new variables $int : fire\_value$ and $boolean : checked$. $fire\_value$ represents its adjacent burning neighbors and *checked* represent it is a checked burning point. The initialed value for them are 0 and false respectively. For detail, please see below.

- F: initial fire points and it will update when there are more vertex satisfy requirements.
- G: The undirected graph
- V: Vertices in G
- E: Edges in G
- $fire\_value$: represent number of current vertex's burning neighbors
- *checked*: represent current burning vertex is checked before

Algorithm 1 $BFS(G, F)$

```
1:  cond ←True
2:  while cond do
3:      cond ←False
4:      for f in F do
5:          if not f.checked then
6:              f.checked = True
7:              for v in G.incident(f) do
8:                  if NOT F[v] then
9:                      v.fire_value+ = 1
10:                 end if
11:             end for
12:         end if
13:     end for
14:     new ←[]
15:     for v in G\F do
16:         if v.fire_value ≥ 3 then
17:             new.append(v)
18:             cond ←True
19:         end if
20:     end for
21:     F.append(new)
22: end while
23: return F
```

Now, I would like to briefly introduce how each line of code works.

- line 1: this assignment just make sure the while loop can at least run one time because we need to at least check the neighbors for the initial burning vertex.
- line 2: this while-loop make sure we can get all vertex that will catch fire in present and future.

- line 3: *cond* will become True when we find at least one vertex will begin burning in current while-loop.
- line 4, 5, 6: In this first for-loop, we will check all vertices that have not been checked before but are currently burning. line 5 + line 6 make sure we only use each burning vertex once. The reason behind that is we do not want to increase $fire\_value$ for the unburning neighbors of burning vertices over and over again.
- line 7 + 8: In this first inner for-loop, I will check current burning vertex's neighbors which are not in F. Since we have an if statement on line 8, we will not access the edges that have already been visited. The reason behind this is that the edges between the current vertex and its burning neighbors have been used to determine why the current vertex began to burn in the previous while-loop.
- line 9: we just set $fire\_value$ for those neighbors. Namely, for vertex in $G \backslash F$, if its $fire\_value \geq 3$, then it will be ignite in current while-loop and we will check it in the second for-loop.
- line 14: array *new* records vertices which will catch fire in current while-loop.
- line 15, 16, 17: In the second for-loop. Firstly, we need to filter vertices which will catch fired in current while-loop. Then we just add those vertices that are in G but not in F to *new*.
- line 18: When we run this code, we have found at least one vertex that will catch fire in the current while loop, so we need to set $cond = True$; Otherwise, we won't be able to detect the rest vertices which may catch fire later. If we did not run this code, which means there will not be anymore vertices catch fire in the future.
- line 21: Add all the vertices that started burning in the current loop to F, and we'll use them in the next while loop.
- line 23: return all burning vertices.

(b)

The idea of my algorithm include 2 steps.

In Step 1, I will add 1 to all the neighbors'$fire\_value$ of current burning vertices f, $where\ f \in F$. Notice that, all burning vertices f have not been checked before (i.e. $f.checked$ should be $False$). Then, go to Step 2.

In Step 2, I will check every vertex v , $where\ v \in G \backslash F$. If $v.fire\_value \geq 3$, then we consider this vertex is the newest burning vertex and we need to add it to F. If we successfully find at least one new burning vertex, then we need to go back to Step 1 to update rest unburning vertex's $fire\_value$ and try to find vertices which will catch fire in the future. If we did not find any vertices catch fire in current step, then we judge that there is no more vertices will catch fire in the future, so the while-loop should stop. Through this description, I think you can clearly identify the correctness of my algorithm.

(c)

The first for-loop will run at most $O(m)$ times throughout whole while-loop, because we will only use each edge e , *where* $e \in E$, once to add the $fire\_value$ for unburning vertex. Specifically,
line 5+6 make sure we will use each burning vertex once. Then, burning vertices' neighbors' $fire\_point$ will increase correctly. Namely, we will not repeatedly use same edges to increase burning vertices' neighbors' $fire\_point$.
line 7+8 make sure we will not access those used edges again in the later while-loop. The reason behind this is that the edges between the current vertex and its burning neighbors have been used to determine why the current vertex burned in the previous while-loop.

The second for-loop will run at most $O(n)$ times throughout whole while-loop. The reason behind it is that there are at most n vertices can be burned. In addition, in each while-loop iteration, we will only access those vertices which start burning in current while-loop. This help us do not repeatedly access other unburning vertices in the future (since some of them will burn in the future, at that time, we may need to access them again if we did not do above step).

Combine them, we get the overall time complexity $O(m + n)$.