SUN Yilin
520030910361

Homework 3

Algorithms
April 6, 2022

1

Algorithm:
We sort the customers by ascending order of their service time, then serve the customer with smallest service time each time.
Proof of Correctness:
The total time can be expressed as

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i} t_j$$

which is simply

$$T = \sum_{i=1}^{n} (n - i + 1) t_i$$

So if the sequence $\{t_1, t_2, t_3, \ldots, t_n\}$ is in ascending order, i.e. $t_1 \leq t_2 \leq t_3 \leq \cdots \leq t_n$, simply by the **rearrangement inequality** we know such a T is the minimum waiting time because it is the so-called "inverse product sum".

Below is the proof of rearrangement inequality for completeness of this problem.
Denote the two sequence that have already sorted by ascending order by $\{a_i\}$ and $\{b_i\}$. Let $p_i = \sum_{j=1}^{i} b_j$, $q_i = \sum_{j=1}^{i} b_{k_j}$. It's easy to see $p_i \leq q_i$ for $i \leq n-1$ and $p_n = q_n$.
Then

$$\begin{aligned}
\sum_{i=1}^{n} a_{n-i+1} b_i &= \sum_{i=1}^{n-1} a_i (p_{n-i+1} - p_{n-i}) + a_n p_1 && (1)\\
&= \sum_{i=1}^{n-1} (a_{n-i+1} - a_{n-i}) p_i + a_1 p_n && (2)\\
&\leq \sum_{i=1}^{n-1} (a_{n-i+1} - a_{n-i}) q_i + a_1 q_n && (3)\\
&= \sum_{i=1}^{n} a_{n-i+1} b_{k_i} && (4)
\end{aligned}$$

Another side of the inequality is not used in our problem, so the proof is omitted here.

2

2.1

Prove by contradiction. Suppose there are two maximal independent sets denoted by A and A' with different size, without loss of generality, assume $|A| < |A'|$. Then by exchange property $\exists x \in A' \backslash A$ such that $A \cup \{x\} \in \mathcal{I}$. This means that there exists $A \cup \{x\} \in \mathcal{I}$ and $A \subset A \cup \{x\}$, which contradicts with the fact that A is a maximal independent set. So maximal independent sets must have the same size.

2.2

We first prove the hereditary property.
Obviously $\mathcal{S}$ is non-empty because at least it contains the empty set. For any $P \in \mathcal{S}$ and any $Q \subset P$, since P is acyclic, Q cannot be acyclic, so $Q \in \mathcal{S}$. So M has the hereditary property.

We then prove the exchange property.
For any two independent sets $A, B \in \mathcal{S}$, without loss of generality, assume $|A| < |B|$. We consider the forest G' induced by A in G. Suppose there are k trees in G' and the number of edges of each tree is denoted by a_i then we have

$$\sum_{i=1}^{k} a_i = |A|$$

Now we consider the "distribution" of edges in B. Since $B \in \mathcal{S}$, B is acyclic. Then the maximum number of edges from B connecting vertices in each tree of G' is at most $a_i, i \in [k]$. But we know $|B| > |A| = \sum_{i=1}^{k} a_i$, which means that there is at least one edge in B connecting one(or two) vertex out of G' or connecting two different trees in G', and we denote this specific edge by e. Then we know $A \cup \{e\}$ is not acyclic. And since it is acyclic, it is in $\mathcal{S}$. So there exists $e \in B \backslash A$ such that $A \cup \{e\} \in \mathcal{S}$. So M has the exchange property.
To sum up, M is a matroid.
The maximal sets of this matroid are all the spanning trees of graph G.

2.3

Prove by contradiction. Suppose there is no maximal independent set with maximum weight containing x, pick an arbitrary maximal set with maximum weight denoted by S and we will show that we can construct another maximal set S' with maximum weight from S.

We will start from S and $\{x\}$. Our condition here is that $x \notin S$ and x has the maximum weight. By exchange property $\exists x_{i_1} \in S$ such that $\{x, x_{i_1}\} \in \mathcal{I}$. Moreover, we can carry out this process as long as $|S| > |\{x, x_{i_1}\}|$. By exchange property we carry out this process $|S| - 1$ times and get a set

$$S' = \{x, x_{i_1}, x_{i_2}, x_{i_3}, \ldots, x_{i_{|S|-1}}\}$$

S' satisfies that $S' \backslash \{x\} \subset S$, $|S'| = |S|$ and $S' \in \mathcal{I}$.
The fact that $|S'| = |S|$ and $S' \in \mathcal{I}$ tells us that S' is a maximal independent set, and the fact that $S' \backslash \{x\} \subset S$, x has maximum weight tells us that $w(S') \geq w(S)$. Since S has maximum weight, S' also has maximum weight. So we have constructed a maximal independent set with maximum weight containing x, which is S'.

2.4

To use this algorithm we only need to formalize our input.
For the MST problem we have a graph $G = (V, E)$ and the matroid is properly defined and proved as shown in problem 2.2, which is $M = (E, \mathcal{S})$. The weight function w is also defined on $E \to \mathbb{R}_{\geq 0}$ in the MST problem. We then use M and w as inputs for our algorithm.
Then this algorithm will pick the element in E with the largest weight as long as $S \cup \{x\} \in \mathcal{I}$, which is equivalent to picking an edge with largest weight as long as the edges does not generate a cycle in Kruskal's algorithm. So after we run this algorithm, we will get a maximal(or minimal) spanning tree.

2.5

We define $\mathcal{S} = \{F \subset U | \text{vectors in } F \text{ are linearly independent}\}$. We now prove $M = \{U, \mathcal{S}\}$ is a matroid and use the algorithm described in problem 2.3 with M and w as input.
Firstly, the hereditary property is true because any subset of a set containing vectors that are linearly independent must also be containing independent vectors.
As for the exchange property, we prove by constadiction.
For any two independent sets $A, B \in \mathcal{S}$, without loss of generality, assume $|A| < |B|$. If exchange property does not hold for A and B, then since A is linearly independent, B can be linearly expressed by A. Since B can be linearly expressed by a set of vertors with size smaller than $|B|$, B itself cannot be linearly independent. So there exists a vector $x \in B$ where $x \notin A$ and $A \cup \{x\} \in \mathcal{S}$.
So M is indeed a matroid. Then as we have stated at the start of this problem, we can run algorithm in 2.3 with M and w and get such a set of linearly independent vectors S with maximum weight.

3

3.1

The idea is pretty straightforward. Since we only need to determine whether B is reachable from A, the gas cost is not to our concern here. So starting from A, we fill up the gas tank as soon as we arrive at a gas station. Following this method, as long as the two adjacent gas stations are within the distance of our tank capacity C, we can always reach the next station from our current station.
Given the inputs, i.e. the distance of these n gas stations, we put them in an array $\boldsymbol{d}$ and sort it by ascending order. Then we calculate the diffenence between adjacent elements in array $\boldsymbol{d}$. If any of the diffenences is larger than C, or if the distance between farthest gas station and D is larger than C, then B is not reachable. If not, then B is reachable.

Algorithm 1 Determine Reachable

```
1:  procedure DETERMINE REACHABLE(d, C, D):        ▷ d = {d_1, d_2, d_3, ..., d_n}
2:      Sort d by ascending order
3:      if D − d_n > C then
4:          return Not Reachable
5:      Initialize i ← 0
6:      while i < n do
7:          l ← d_{i+1} − d_i
8:          if l > C then
9:              return Not Reachable
10:     return Reachable
```

Proof of Correctness:
The algorithm is correct because, as is described above, it always guarantees that the reachability of next station can be determined by the nearest gas station before it. Thus if all the gas stations are reachable from the previous stations and D is reachable from the farthest station, B is reachable from A.
Analysis of Running Time:
The time complexity of the sort procedure at line 2 is $O(n \log n)$. After we sort the sequence, the while loop will finish with time $O(n)$. So the total running time is $O(n \log n)$.
If the input in $\boldsymbol{d}$ is given us ordered, the total running time is just $O(n)$.

3.2

Since the total distance D is definite, the total amount of gas needed is D. To be specific, we do not have to pre-fuel the tank to reach B.
So the idea is to always use the cheapest gas possible. We put the price of n gas stations and put them in an array $\boldsymbol{p}$, then we sort $\boldsymbol{p}$ in a ascending order. We pick the cheapest p_i each time and calculate the interval on which we can use the gas from i-th station to travel. A detailed algorithm is below:

Algorithm 2 Minimal Cost

procedure MINIMAL COST($\boldsymbol{p}, \boldsymbol{d}, C, D$):
    Make pair of elements in $\boldsymbol{p}$ and $\boldsymbol{d}$ for sort
    Sort $\boldsymbol{p}$ by ascending order of p_i
    Initialize an empty interval $L = \emptyset$ ▷ Update L at each iteration
    Initialize $c \leftarrow 0$ ▷ c is the total cost
    Initialize $i \leftarrow 0$
    while $i \leq n$ **do**
        calculate a closed interval $L_i = [d_i, d_i + C] \cap [0, D]$
        calculate the length l_i of $L_i \backslash L$ ▷ $\mathrm{O}(\log n)$ with binary search tree
        $c \leftarrow c + p_i l_i$ ▷ Update the total cost
        $L \leftarrow L \cup L_i$ ▷ $\mathrm{O}(\log n)$ by maintaining a binary search tree
        if $L = [0, D]$ **then**
            return c
    return -1

Actually this algorithm can also help us determine whether B is reachable. If the result given by this algorithm is negative, then B is not reachable.

Proof of Correctness:

Suppose there is another gas expenditure plan with minimal total cost, we will show that our algorithm will provide a solution at least better.
We can assume that our car can sort of "remember" to use gas from which gas station on any of these intervals. Or to be specific, on the interval $Li \backslash L$ of each iteration, we assume our car is using gas from the i-th gas station. Under this definition, we prove our "gas plan" is at least better than another optimal gas plan by induction.
In this induction, we construct our plan P_n from P^* and show that P_i is a improvement over P^* at each iteration.
Let $P_0 = P^*$ and we then do induction.
Outside interval $L_1 \backslash L$, we use plan P_0, on interval $L_1 \backslash L$, we use our plan P_n, we update P_1 by combining the plans at P_0 and P_n on the whole interval $[0, D]$. then obviously P_1 is better

than $P_0 = P^*$ because at least it is better on $L_1 \backslash L$.
We now construct P_k from P_{k-1}:
Outside interval $L_k \backslash L$, we use plan P_{k-1}, on interval $L_k \backslash L$, we use our plan P_n, Then at each iteration our P_k is at least an improvement on interval $L_k \backslash L$ while the cost outside $L_k \backslash L$ remain unchanged. So after the iterations finally our P_n is better than $P_0 = P^*$. Since P^* is optimal, our P_n is also optimal.

Analysis of Running Time:

The time complexity of our algorithm comes from line 3, 9 and 11.
The time complexity of line 3 is $O(n \log n)$ for the sort process, and for the operations at line 9 and 11, if we maintain a binary search tree with nodes representing the intervals already calculated. we can finish each iteration with time $O(\log n)$. This is possible because, given the properties of the intervals added are all of the same length C, we will be combining at most three intervals together each time. So the total time for while loop is $O(n \log n)$.
Thus the total time complexity of our algorithm is $O(n \log n)$.

4

4.1

The idea is to initialize a root r with degree larger or equal to 2, make the parent of r itself, then do BFS from r and calculate the depth of all the vertices. After this find the deepest leaf and find its parent p.
Then do BFS from p for one step to mark all the vertices that is connected to p as covered. At each iteration, we pick a single parent p and mark some vertices.
Then we find the deepest leaf that has not been covered and repeat the process until all the vertices are covered in the graph.

Algorithm 3 1-Minimal Cover

1: **procedure** 1-Minimal Cover($G = (V, E)$): ▷ G must be a tree
2: Initialize set of vertices S, S is empty
3: Initialize an arbitrary root r of G, let parent$(r) = r$ ▷ deg$(r)\geq 2$ for convenience
4: BFS(r) to get the depth of all vertices in V
5: **while** V is not totally covered **do**
6: $p \leftarrow$ Find parent of the **deepest uncovered** leaf in G ▷ O(1) by "depth" vector
7: $S \leftarrow S \cup \{p\}$
8: Mark all vertices connected to p as already covered. ▷ BFS for one step
9: **return** S

Proof of Correctness:
We prove by showing that our choice of vertices is necessary at each iteration and is global optimal as well.
For convenience of analysis, suppose at current iteration $|V| > 2$, we know that an optimal minimal cover set must cover all the leaves in G, and we also know that the deepest uncovered leaf can ony be covered by picking itself or its parent. But obviously the choice of parent is better than picking the leaf itself because the parent can cover at least the same set of vertices as its child. So our choice of parents is indeed necessary at each iteration.
The reason why the update process at Line 8 does not violate global optimal is that once a vertex is covered, we do not have to consider it ever again. So again we find the deepest uncovered leaf and carry out the same procedure. The reason why we choose the deepest leaf is not explained here because for $k = 1$ it is not that obvious why we must choose the deepest one each iteration. But we will show that for general k this algorithm will not perform well if not choosing the deepest leaf.
To sum up, our choice of vertices at each iteration must be necessary, or equivalently and

more specificly, for any optimal cover set for this tree, at least all the vertices chosen by our algorithm must be in this optimal set so it can generate the minimal cover set of G.

Analysis of Running Time:

Since $|E| = |V| - 1$, we can describe the time complexity only by $|V|$. The time spent by our algorithm that is not constant time only comes from BFS at line 4 and while loop at line 5. The BFS at line 4 will traverse the whole tree once and the while loop will only mark all the vertices once. As for the BFS at line 8, it will be carried out within constant time because we only do BFS for one step. So the algorithm kind of traverse the graph twice, causing a time complexity of $O(|V|)$.

4.2

The idea is similar to the case with $k = 1$ and we only need to make some subtle changes to our algorithm. But pay attention that this time the choice of the deepest uncovered leaf at each iteration is crucial to the correctness of our algorithm.
The first change is that we do not pick the parent of a leaf. We still need to find all the deepest leaf, but we are going to pick the ancestor p which has a path with exactly length k of the leaf. Then we do BFS from p for k steps and mark all the vertices that is within distance k with p as covered. Similarly at each iteration we find an ancestor and mark some vertices.
The other change is how we exit the procedure. We cannot guarantee that after some iterations the deepest uncovered leaf still has an ancestor with exactly distance k So we pick the root when the depth of deepest uncovered leaf is smaller or equal to k and end the procedure.

Algorithm 4 k-Minimal Cover

```
1:  procedure k-Minimal Cover(G = (V, E)):                ▷ G must be a tree
2:      Initialize set of vertices S, S is empty
3:      Initialize an arbitrary root r of G               ▷ deg(r)≥2 for convenience
4:      BFS(r) to get the depth of all vertices in V
5:      while V is not totally covered do
6:          l ← the deepest uncovered leaf
7:          if the depth of l is smaller or equal to k then
8:              S ← S ∪ {r}
9:              return S
10:         p ← Find ancestor with exactly distance k of l
11:         S ← S ∪ {p}
12:         Mark all vertices within distance k of p as already covered
13:     return S
```

Proof of Correctness:
Similar to our proof with $k = 1$, at each itertaion, our choice of the ancestor p is necessary because we still need to cover the current deepest leaf. And to cover this leaf, using the ancestor p with exactly distance k is the best choice because p is capable of covering any other vertices that can be covered by p's descendants. But we cannot choose any ancestor of p because that will cause current deepest leaf remaining uncovered.
The reason why we choose the deepest uncovered leaf is revealed here: Only by choosing the deepest one can we guarantee that this leaf cannot be covered by other leaves' ancestors

chosen in this way.

Analysis of Running Time:

The time complexity of a general k is a little bit different. Although in the case $k = 1$ any vertex will only be visited constant times, in general case there is no guarantee that a vertex will not be visited many times.

because we still only need to traverse some vertex each time if the tree is ill-shaped. But after each iteration the number of vertices reduced is at least k because the parent p has at least k descendants.

Thus the running time can be expressed as

$$|V| + (|V| - k) + (|V| - 2k) + \cdots + 1$$

So the total time complexity is still $O(\frac{|V|^2}{k})$.

5 Comments

5.1

About two whole days excluding time spent on sleeping and eating(and loafing perhaps).....

5.2

Actually the most difficult question from my perspective is problem 2 about matroid. I do not think I will be able to work out this problem without the hint. The hint actually reduces the time spent on thinking largely. But I will still give this problem a big 5.

Problem 4 will also get a 5. Although the intuition of this problem is not that hard to find, the implement for a formal algorithm surely takes some time. Even if you finish your algorithm, you might still need to modify it several times. Moreover, the running time analysis also takes a lot of work.

As for problem 3 the algorithm is not that hard to figure out, what takes time is how to prove the correctness and analyze the running time. So my recommendation is 4.

Problem 1 is down-right application of rearrangement inequality, so the rating is 2. If the proof of rearrangement inequality is not required, the rating can be reduced to 1.

5.3

No collaborators this time.(Big thanks to the hint.)