SUN Yilin
520030910361

Homework 3

Algorithms
April 5, 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 $\{t_1,t_2,t_3,\dots,t_n\}$ is in ascending order, i.e. $t_1\le t_2\le t_3\le\cdots\le t_n$, 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 $p_i\le q_i$ for $i\le n-1$ and $p_n=q_n$.
Then

$$\begin{aligned}\sum_{i=1}^{n}a_{n-i+1}b_i&=\sum_{i=1}^{n-1}a_i(p_{n-i+1}-p_{n-i})+a_np_1 &(1)\\&=\sum_{i=1}^{n-1}(a_{n-i+1}-a_{n-i})p_i+a_1p_n &(2)\\&\le\sum_{i=1}^{n-1}(a_{n-i+1}-a_{n-i})q_i+a_1q_n &(3)\\&=\sum_{i=1}^{n}a_{n-i+1}b_{k_i} &(4)\end{aligned}$$

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

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

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($\boldsymbol{d}, C, D$):        ▷ $\boldsymbol{d} = \{d_1, d_2, d_3, \ldots, d_n\}$
 2:     Sort $\boldsymbol{d}$ by ascending order
 3:     if $D - d_n > C$ then
 4:         return Not Reachable
 5:     Initialize $i \leftarrow 0$
 6:     while $i < n$ do
 7:         $l \leftarrow d_{i+1} - d_i$
 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 $O(n \log n)$.

3.2

4

4.1

The idea is to find the leaves and pick all of their parents and delete the vertices that is connected to the parents picked each time, then repeat the process until there is no more vertices in the original graph. If the graph is left with a single root after some iterations of the process, we also pick the root. (If not, the root vertex must have been deleted when we picked one of its children)
A detailed algorithm is below:

Algorithm 2 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:      while $V$ is not empty do
4:          if $|V| = 1$ then
5:              $S \leftarrow S \cup V$
6:              return $S$
7:          $P \leftarrow$ Find parents of all the leaves in $G$     ▷ by DFS procedure
8:          $S \leftarrow S \cup P$
9:          Update $G$ by deleting neighbourhood of $P$ and $P$ themselves
10:     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.
If $|V| = 1$, then our algorithm is trivally true.
Suppose at current iteration $|V| > 1$, we know that an optimal minimal cover set must cover all the leaves in G, and we also know that the leaves can ony be covered by picking themselves or their parents. But obviously the choice of parents are better than picking the leaves themselves because for any pick of parent, 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 9 does not violate global optimal is that once a vertex is covered, we do not have to consider it ever again. So we can immediately delete the vertices that are covered by the parents each iteration and reduce our problem to a similar one with smaller scale.
To sum up, our choice of vertices must be necessary, 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 the while loop and the DFS at line 7. At line 7 the DFS process must visit each vertex and edge at least once, causing a time complexity of $O(|V|)$ each time. But note that after each iteration the size of $|V|$ is slightly reduced by the updating process, in the worst case the updating process only deletes 3 vertices from V, so the worst time complexity is

$$|V| + (|V| - 3) + (|V| - 6) + \cdots + 1$$

which is $O(|V|^2)$.

4.2

The idea is similar to the case with $k = 1$ and we only need to make some subtle changes to our algorithm.
The first change is that we do not pick the parents of all the leaves. We still need to find all the leaves, but we are going to pick the ancestors which have a path with exactly length k of all those leaves. And we delete the ancestors that are within distance k of these ancestors picked at each iteration. The descendants are deleted as well.
Another change is that we do not end by picking the only root if it remains in the original graph. Instead, we pick the root when the depth of remaining sub-tree is smaller or equal to k and end the procedure.

Algorithm 3 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:     while $V$ is not empty do
4:         if the depth of $G$ is smaller or equal to $k$ then
5:             $S \leftarrow S \cup \{r\}$                          ▷ $r$ is the root of tree
6:             return $S$
7:         $P \leftarrow$ Find ancestors with exactly distance $k$ of all the leaves in $G$
8:         $S \leftarrow S \cup P$
9:         Update $G$ by deleting ancestors and descendants within distance $k$ of $P$
10:    return $S$
```

Proof of Correctness:
Similar to our proof with $k = 1$, at each itertaion, our choice of the ancestors is necessary because we still need to cover all the leaves. And to cover the leaves, finding the ancestors

with exactly distance k is the best choice because they will cover any vertices if we choose another set of vertices.

Analysis of Running Time:

Similar to our analysis with $k = 1$, now our total time is

$$|V| + (|V| - 2k - 1) + (|V| - 4k - 2) + \cdots + 1$$

which is $O(\frac{|V|^2}{k})$

5