# Algorithm Homework 03

Qiu Yihang

April 2022

## 1 Problem 01

*Solution.* We design algorithm as follows.

**Algorithm 1:** Optimal Order to Process Customers

**Function** *Optimal Order* ($t[1, 2, ..., n]$)
  $index \leftarrow [1, 2, ..., n]$;
  sort($t, index$);
  // Sort $t$ and $index$ by $t$ from the smallest to the largest. After the sort, $index[\cdot]$ stores the index of the customer whose service time is $t[\cdot]$.
  **Return:** $index$
**end**

In short, we serve customers in order of increasing service time $t_i$.

Now we prove the correctness of the algorithm.

Let our order be $k_1, k_2, ..., k_n$. Define $\tau_i \triangleq t_{k_i}, A_i \triangleq \sum_{j=1}^{i} \tau_j$. Then $\tau_1 \leq \tau_2 \leq ...\tau_n$.

The total waiting time is

$$\hat{T} = \sum_{i=1}^{n} \sum_{j=1}^{i} t_{k_j} = \sum_{i=1}^{n} A_i$$

For any order $m_1, m_2, ..., m_n$, define $B_i \triangleq \sum_{j=1}^{i} t_{n_j}$. The total waiting time is

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i} t_{m_j} = \sum_{i=1}^{n} B_i$$

Now we prove $\forall i, A_i \leq B_i$ by contradiction. Assume $\exists q$ s.t. $A_q > B_q$. Then $\tau_q > t_{m_q}$.

Since $\tau_1 \leq \tau_2 \leq ... \leq \tau_n$, we know $\tau_1 \leq \tau_2 \leq ... \leq \tau_{q-1} \leq t_{m_q} < \tau_q \leq \tau_{q+1} \leq ...\tau_n$,

i.e. $t_{m_q} \notin \{\tau_1, \tau_2, ...\tau_n\} = \{t_{m_1}, t_{m_2}, ...t_{m_n}\}$. **Contradiction.**

Therefore, we have

$$T = \sum_{i=1}^{n} B_i \geq \sum_{i=1}^{n} A_i = \hat{T}$$

Thus, the order given by our algorithm is the optimal order with minimum total waiting time. ■

# 2 Problem 02

## 2.1 In a Matroid, Maximal Independent Sets are of the Same Size

*Proof.* We prove maximal independent sets in $M$ are of the same size by contradiction.

Suppose exist two maximal independent sets $A, B$ in $M = (U, \mathcal{I})$. Then $A, B \in \mathcal{I}$. Without loss of generality, assume $|A| > |B|$.

Since $B$ is a maximal independent set, there is no $C \in \mathcal{I}$ s.t. $B \subsetneq C$. Meanwhile, by **Exchange Property**, we know $\exists\ x \in A \setminus B$ s.t. $B \cup \{x\} \in \mathcal{I}$, i.e. exists an independent set $\widetilde{B} = B \cup \{x\} \in \mathcal{I}$ s.t. $B \subsetneq \widetilde{B}$. **Contradiction**.

Thus, maximal independent sets are of the same size. ■

## 2.2 Matroids on a Simple Undirected Graph

*Solution.* First we show that $M = (E, \mathcal{S})$ is a matroid,

i.e. to prove $M$ has possess hereditary property and exchange property.

**Proof of Hereditary Property.**

Obvious $\varnothing \in \mathcal{S}$, i.e. $\mathcal{S}$ is non-empty.

For any $A \in \mathcal{S}$, by the definition of $\mathcal{S}$, we know $A \subset E$ and $A$ is acyclic. Since removing edges from an acyclic graph still gives an acyclic graph, we know $\forall B \subset A$, $B$ is acyclic, i.e. $B \in \mathcal{S}$.

Thus, for every $A \in \mathcal{S}$ and $B \subset A$, it holds that $B \in \mathcal{S}$. □

**Proof of Exchange Property.**

For every $A, B \in \mathcal{S}$ with $|A| < |B|$, we know exists $x = (u, v) \in B$ s.t. $u$ and $v$ are not connected in $A$. (Otherwise, $\forall u, v$ s.t. $u$ and $v$ are connected in $B$, $u$ and $v$ are also connected in $A$. Then $B$ must contain a cycle since at most $|A|+1$ vertices are adjacent to at least an edge in $B$, while the number of edges $|B| \geq |A| + 1$.)

Therefore, $A \cup \{(u, v)\}$ does not contain a cycle, i.e. $A \cup \{x\} = A \cup \{(u, v)\} \in \mathcal{S}$. Moreover, since $u$ and $v$ are not connected in $A$, $(u, v) \notin A$, i.e. $x = (u, v) \in B \setminus A$.

Thus, for any $A, B \in \mathcal{S}$, exists some $x \in B \setminus A$ s.t. $A \cup \{x\} \in \mathcal{S}$. □

In conclusion, $M$ is a matroid. ■

By the definition of $\mathcal{S}$, we know all independent sets in $M$ can induce to forests in $G$. Thus, $\forall A \in \mathcal{S}, |A| \leq |V| - 1$. Obvious any spanning tree $T \in \mathcal{S}$ while $|T| = |V| - 1$. Thus, the maximal sets of this matroid are **all spanning trees** of $G$. ■

## 2.3 Correctness of the Given Algorithm

*Proof.* From the algorithm, we know $\{x\} \in \mathcal{I}$ and $\forall S' \in \mathcal{I}, \forall y \in S', w(x) \geq w(y)$.

Now we prove that there exists a maxmimal independent set $S' \in \mathcal{I}$ with maximum weight containing $x$ by contradiction.

**Assumption.** Assume all maximum independent sets with maximum weight do not include $x$.

We choose an arbitrary maximum independent set $S$ with maximum weight. By hereditary property, we know $\forall A \subset S, A \in \mathcal{I}$.

Define $X_1 \triangleq \{x\}$. We already know $X_1 \in \mathcal{I}$.

Pick $A_2 \subset S$ s.t. $|A_2| = 2$, we know $A_2 \in \mathcal{I}$.

By exchange property, exists some $y_2 \in A_2 \setminus X_1$ s.t. $X_2 \triangleq X_1 \cup \{y_2\} \in \mathcal{I}$.

We have $|X_2| = 2$.

Pick $A_3 \subset S$ s.t. $|A_3| = 3$, we know $A_3 \in \mathcal{I}$.

By exchange property, exists some $y_3 \in A_3 \setminus X_2$ s.t. $X_3 \triangleq X_2 \cup \{y_3\} \in \mathcal{I}$.

We have $|X_3| = 3$.

...

Pick $A_{|S|} = S \subset S$ s.t. $|A_{|S|}| = |S|$, we know $A_{|S|} \in \mathcal{I}$.

By exchange property, exists some $y_{|S|} \in A_{|S|} \setminus X_{|S|-1}$ s.t. $X_{|S|} \triangleq X_{|S|-1} \cup \{y_3\} \in \mathcal{I}$.

We have $\left|X_{|S|}\right| = |S|$, i.e. $X_{|S|}$ is a maximal independent set.

Now we consider $X_{|S|}$ and $S$. Obvious exists exactly one $y_0$ s.t. $y_0 \in S, y_0 \notin X_{|S|}$.

Therefore, we have

$$\begin{aligned} w(X_{|S|}) &= w(x) + \sum_{y \in X_{|S|},\ x \neq y} w(y) \\ &= w(x) + \sum_{y \in S\ \wedge\ y \in X_{|S|}} w(y) \\ &\geq w(y_0) + \sum_{y \in S\ \wedge\ y \in X_{|S|}} w(y) \\ &= \sum_{y \in S} w(y) = w(S). \end{aligned}$$

Meanwhile, by the assumption, we know $w(S) > w(X_{|S|})$. **Contradiction.**

Thus, there must be a maximal independent set $S' \in \mathcal{I}$ with maximum weight containing $x$. ■

## 2.4 MST Case

*Solution.* We can solve MST using algorithm in **2.3** as follows.

Let the graph be $G = (V, E, \texttt{weight})$. we define matroid $M = (E, \mathcal{S})$, where $\mathcal{S} = \{F \subset E \mid F \text{ is acyclic}\}$.

$w : E \mapsto \mathbb{R}^*$ is defined as follows.

Let $w_{max} = \max\limits_{(u,v)\in E} \texttt{weight}\left((u, v)\right)$. We define $w\left((u, v)\right) = w_{max} - \texttt{weight}\left((u, v)\right) \geq 0$.

Then we apply the algorithm in **2.3** on $M$ and $w$. The output of the algorithm is a Minimum Spanning Tree of $G$. ■

The reason that it is equivalent to Kruskal's Algorithm is as follows.

In **Kurskal's Algorithm**, we select edges in order of increasing weight of edge. Also, we only pick the edge when the addition of the edge won't cause a cycle. Since selecting edges in order of increasing `weight` is exactly picking edges in order of decreasing $w$, and [the addition of the edge won't cause a cycle] $\iff S \cup \{x\} \in \mathcal{S}$, we know our application of the algorithm in **2.3** is equivalent to **Kurskal's Algorithm**. ■

## 2.5 Algorithm Design

*Solution.* First, we construct a matroid as follows.

We define $\mathcal{I} = \{F \subset U \mid \text{all vectors in } F \text{ are linearly independent to each other}\}$, $M = (U, \mathcal{I})$.

Now we prove $M$ is a matroid.

**Proof of Hereditary Property.**

Obvious $\forall \mathbf{x} \in U$, $\{\mathbf{x}\} \in \mathcal{I}$, i.e. $\mathcal{I}$ is non-empty.

For any $A \in \mathcal{I}$ and $B \subset A$, obvious all vectors in $A$ are linearly independent, which gives that all vectors in $B$ are linearly independent, i.e. $B \in \mathcal{I}$. □

**Proof of Exchange Property.**

By contradiction. Assume exist $A, B \in \mathcal{I}$ with $|A| < |B|$ s.t. $\forall \mathbf{x} \in B \setminus A,\ A \cup \{\mathbf{x}\} \notin \mathcal{I}$.

Thus, any vector in $B$ is linearly dependent to some vectors in $A$, i.e. $\texttt{dim}\left(\texttt{span}(B)\right) \leq \texttt{dim}\left(\texttt{span}(A)\right) \leq |A| < |B|$. This gives that $B$ contains at most $|A|$ linearly independent vectors, i.e. exists at least one vector in $B$ that is linearly dependent to another vector in $B$. **Contradiction** to the assumption that $B \in \mathcal{I}$. □

Therefore, $M$ is a matroid.

Thus, we can **apply the algorithm in 2.3 on $M$ and $w$**. The result is exactly the set $S \subset U$ with maximum weight and all vectors in $S$ are linearly independent. ■

# 3 Problem 03

## 3.1 Reachablity

*Solution.* Let $d_{n+1} = D$. We call $B$ the $(n+1)$-th gas station.

Without loss of generality, we assume $d_1 = 0 \leq d_2 \leq ... \leq d_n \leq d_{n+1} = D$.

Obvious, if $B$ is reachable from $A$ with tank capacity $C$, any $(i+1)$-th gas station is reachable from the $i$-th station with at most $C$ units of gas ($i = 1, 2, ..., n$), i.e. the distance between any $i$-th station and $(i+1)$-th station is no larger than $C$.

Therefore, $\forall i \in \{1, 2, ..., n\}, d_{i+1} - d_i \leq C \Longleftrightarrow B$ is reachable from $A$.

Thus, to check the reachability from $A$ to $B$, we just need to check whether all $i \in \{1, 2, ..., n\}$ satisfy $d_{i+1} - d_i \leq C$. (Note that by our definition, $d_{n+1} = D$.) ■

Based on the idea above, we design the following algorithm.

**Algorithm 2:** Reachability Check

**Function** *Reachability Check* $(d[1, 2, ..., n], D, C)$
  $sort(d)$;
  `// Sort in order of increasing` $d_i$`. If` $d_i$ `is already sorted, this step should be ommited.`
  **if** $D - d[n] > C$ **then Return:** `False`;
  **for** $i = 1 \to n - 1$ **do**
    **if** $d[i+1] - d[i] > C$ **then Return:** `False`;
  **end**
  **Return:** `True`
**end**

The proof of the correctness of the algorithm is given above. ■

Now we analyze the time complexity of our algorithm.

Note that the problem does not guarantee that $d$ is sorted. Thus, since $sort(\cdot)$ takes at least $O(n \log n)$ time, the time complexity is $O(n \log n)$. If the given $d$ is sorted, the time complexity is $O(n)$.

Thus, the time complexity of our algorithm is

$$T(n) = \begin{cases} O(n \log n), & \text{the given } d \text{ is not sorted.} \\ O(n), & \text{the given } d \text{ is sorted.} \end{cases}$$

■

### 3.2 Minimum Gas Cost

*Solution.* Our strategy is as follows.

When we arrived at gas station $i$, we first ensure that we can make it to the next station.

If the gas in the tank is not enough, refuel it so that we can get to the next station.

Then we see if refueling more gas here is better.

We keep refueling gas until the tank is full, or the gas in the tank can support us arrive at a station where gas is cheaper than it is here.

Based on the idea above, we design the algorithm as follows.

**Algorithm 3:** Minimum Gas Cost

**Function** *Minimum Gas Cost* $(d[1, 2, ..., n], D, p[1, 2, ..., n], C)$
- $sort(d, p)$; `// Sort` $d, p$ `in order of increasing` $d_i$
- $cost \leftarrow 0, dist \leftarrow 0$;
- $d \leftarrow d \cup \{D\}$, i.e. $d[n+1] \leftarrow D$;
- `//` *cost* `is the minimum cost on gas to reach` *dist* `distance away from` $A$.
- $j \leftarrow 0$;
- **for** $i = 1 \rightarrow n$ **do**
  - `// We arrived at the i-th gas station.`
  - `// First, we need to make sure that we can make it to the next station.`
  - $tank \leftarrow dist - d[i]$;
  - **if** $dist < d[i+1]$ **then** $tank \leftarrow d[i+1] - d[i]$;
  - `// Then we see if refueling more gas here is cheaper than refueling gas later at other stations.`
  - **if** $tank > C$ **then Return:** $-1$;
  - **while** $(tank < C) \wedge (j \leq n+1)$ **do**
    - **if** $p[i] > p[j]$ **then Break**;
    - **if** $d[j] - d[i] > C$ **then**
      - $tank \leftarrow C$;
      - **Break**;
    - **end**
    - $tank \leftarrow d[j] - d[i]$;
    - $j \leftarrow j + 1$;
  - **end**
  - $cost \leftarrow cost + p[i] \cdot (d[i] + tank - dist)$;
  - $dist \leftarrow d[i] + tank$;
  - **if** $dist = D$ **then Break**;
- **end**
- **Return:** *cost*

**end**

Now we prove the correctness of the algorithm.

We use $\mathcal{S}$ to represent different refueling strategies. Suppose moving $x$ from $x+1$ is supported by a unit of gas refueled at $\mathcal{S}[x]$, and we won't use gas refueled at station $i$ until gas refueled at station $1 \sim (i-1)$ is all consumed, i.e. $\forall x, y \in \{0, 1, ..., D\}, x < y \Rightarrow \mathcal{S}[x] \leq \mathcal{S}[y]$.

We use $\texttt{cost}(\mathcal{S})$ to denote the gas cost of strategy $\mathcal{S}$, i.e. $\texttt{cost}(\mathcal{S}) = \sum_{i=0}^{D} p[\mathcal{S}[i]]$.

Let the strategy computed by our algorithm be $S^{\star}$. Now we prove $S^*$ is the best strategy, i.e. the strategy with minimum gas cost, by contradiction.

Assume $\min\limits_{\mathcal{S}} \texttt{cost}(\mathcal{S}) < \texttt{cost}(S^*)$. Let $S = \operatorname{argmin}\limits_{\mathcal{S}} \texttt{cost}(\mathcal{S})$.

We can always find the smallest $x$ s.t. $S[x] \neq S^*[x]$. By the process of our algorithm, we know $S^*[x] < S[x],\ p[S^*[x]] \leq p[S[x]]$. Otherwise, $S[x]$ will cause the tank to overflow.

**CASE 1.** $p[S^*[x]] < p[S[x]]$.

Then we can construct a strategy $S'$ as follows.

$$S' = \begin{cases} S^*[i] & i \leq x \\ S[i] & i > x \end{cases}$$

Since for $\forall i < x, S[i] = S^*[i] \leq S^*[x]$ and for $\forall i > x, S[i] > S[x] > S^*[x]$, at any time, the tank in strategy $S'$ has no more gas than in strategy $S$ or $S^*$. Therefore, $S'$ is a valid strategy.

Obvious $\texttt{cost}(S') = \texttt{cost}(S) - p[S[x]] + p[S^*[x]] < \texttt{cost}(S)$. Contradiction to the assumption $S$ is the strategy with the minimum gas cost.

Thus, $p[S[x]] \leq p[S^*[x]]$. **Contradiction** to the assumption that $S = \operatorname{argmin}\limits_{\mathcal{S}} \texttt{cost}(\mathcal{S})$.

**CASE 2.** $p[S^*[x]] = p[S[x]]$.

We know $\texttt{cost}(S[1:x]) = \texttt{cost}(S^*[1:x])$. Then we repeat the process above on strategy $S[(x+1):D]$ and $S^*[(x+1):D]$.

If $p[S^*[x]] = p[S[x]]$ holds for any $x \in \{0, 1, ..., D\}$ s.t. $S^*[x] \neq S[x]$,

we have $\forall x \in \{0, 1, ..., D\}, p[S^*[x]] = p[S[x]]$, i.e.

$$\texttt{cost}(S^*) = \texttt{cost}(S) = \min_{\mathcal{S}} \texttt{cost}(\mathcal{S}) < \texttt{cost}(S^*).$$

**Contradiction**.

Therefore, our algorithm can give the minimum gas cost. ■

# 4 Problem 04

## 4.1 $k = 1$.

*Solution.* It is trivial that for any leaf vertex of a tree, selecting its parent is better than selecting itself, since its parent can cover at least two vertices while itself can only cover exactly two vertices.

Thus, we design the following algorithm to repetitively find the leaves of the uncovered part of the tree and adding its parent into the covering subset.

**Algorithm 4:** 1-Influential Minimal-Size Cover

**Function** *Minimal Cover* $(G)$
- `//` $G = (V, E)$. `Also,` $G$ `must be a tree.`
- $G' \leftarrow G,\ S \leftarrow \varnothing$;
- Choose a reasonable vertex as the root of $G$;
- **while** $G$ *is not fully covered,* i.e. $G'$ *is not empty* **do**
  - Find a deepest leaf vertex $u$;
  - Find the parent of $u$, i.e the only vertex $v$ s.t. $\{u, v\} \in E$;
  - **if** *cannot find such* $v$ **then**
    - $S \leftarrow S \cup \{u\}$;
    - **Break**;
  - **end**
  - $S \leftarrow S \cup \{v\}$;
  - Delete vertices covered by $v$ from $G'$, i.e. delete $v$ and all vertices adjacent to $v$;
- **end**
- **Return:** $S$

**end**

Now we prove the correctness of the algorithm by induction. Let $S$ be the minimum-size subset of vertices that covers all the vertices in a tree $G$.

We prove that each step is necessary to reach a global optimal solution.

If $|V| = 1$, obvious $S = V$. This is exactly the solution given by our algorithm. Now we consider the case when $|V| \geq 2$.

For any leaf $u$ of $T$, it is obvious that we should choose the only vertex $v$ adjacent to $u$, i.e. the parent of $u$, to be in $S$. This is because $u \in S$ can only cover exactly two vertices while $v \in S$ can cover at least two vertices.

For any vertex $u$ with all its children covered and not in $S$, choosing its parent $v$ is necessary as long as such $v$ exists. (If $v$ does not exist, then selecting $u$ itself is the only choice.) If choose $u$ itself, other vertices adjacent to $v$ will not be covered and thus need at least one extra vertex in $S$. If choose $u$'s children, $v$ is not covered and need at least one extra vertex in $S$. Thus, $v \in S$ is necessary for global optimal $S$.

Therefore, our algorithm will return a correct answer. ■

Now we analyze the time complexity of the algorithm.

Given that $G$ is a tree, we know $|E| = |V| - 1$. The degree of all vertices in $G$ only need to be calculated at the beginning since in the later process, finding vertices with 1 degree can be done in the process of deletion. Therefore, all vertices will be visited only constant times.

Thus, the time complexity is $\underline{\boldsymbol{O(|V|)}}$. ■

## 4.2 General Case

*Solution.* The basic idea is similar to the case when $k = 1$.

The only adjustment from the algorithm when $k = 1$ is that each time we find a deepest leaf of the uncovered part of the tree, we select its ancestor with exactly $k$ edges away from it. If such ancestor does not exist, we select its farthest ancestor.

**Algorithm 5:** $k$-Influential Minimal-Size Cover

```
Function Minimal Cover (G, k)
    // G = (V, E).  Also, G must be a tree.
    G' ← G, S ← ∅;
    Choose a reasonable vertex as the root of G;
    while G is not fully covered, i.e. G' is not empty do
        Find a deepest leaf vertex u;
        Find the ancestor of u, s.t. the distance from u to v is exactly k;
        if cannot find such v then
            Find the farthest ancestor w of u;
            S ← S ∪ {w};
            Break;
        end
        S ← S ∪ {v};
        Delete vertices covered by v from G', i.e. delete v and all vertices within k edges from v;
    end
    Return: S
end
```

Now we prove the correctness of the algorithm. Let the minimum-size subset covering all vertices in $G$ be $S$.

By the process of our algorithm, each time we pick a vertex $u$, then $u$ is covered while all children of $u$ are covered and are not in $S$. In this case, selecting the ancestor $k$ edges away from $u$ (if exists) is necessary for $S$ to be the minimum-size set covering all vertices. Let the ancestor be $w$. Selecting $w$ can always cover all the vertices covered when selecting any children of $w$ to be in $S$. Meanwhile, under most circumstances, $w$ can cover more vertices, such as the ancestor of $w$ which is $k$ edges away. If such $w$ does not exists, we select the farthest ancestor $w$, which still holds the same property. Therefore, each step of our algorithm is necessary for $S$ to be the optimal solution.

Now we analyze the time complexity of the algorithm.

Finding the deepest leaf vertex is to find the vertex with 1 degree in $G'$ and the maximum height. By calculating the height of all vertices in $G'$ at the beginning of the algorithm while updating the degree of all vertices in the process of deletion, we can find such vertex during the process of deletion, by adding vertices of 0 degree into a queue. Thus, for each iteration, the deletion and degree update takes $O(|V|)$. Therefore, the time complexity of the algorithm is $\underline{\boldsymbol{O(|V|^2)}}$. ■

# 5 Rating and Feedback

The completion of this homework takes me six days, about 33 hours in total. Still, writing a formal solution is the most time-consuming part.

The ratings of each problem is as follows.

| Problem | Rating |
|---|---|
| 1 | 2 |
| 2.1 | 1 |
| 2.2 | 2 |
| 2.3 | 2 |
| 2.4 | 2 |
| 2.5 | 2 |
| 3.1 | 1 |
| 3.2 | 3 |
| 4.1 | 3 |
| 4.2 | 4 |

Table 1: Ratings.

This time I finish all problems on my own.