Lab03-GreedyStrategy

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2020.

∗ If there is any problem, please contact TA Shuodian Yu.
∗ Name:Zehao Wang Student ID:518021910976 Email: davidwang200099@sjtu.edu.cn

1. There are $n+1$ people, each with two attributes $(a_i, b_i), i \in [0, n]$ and $a_i > 1$. The i-th person can get money worth $c_i = \frac{\prod_{j=0}^{i-1} a_j}{b_i}$. We do not want anyone to get too much. Thus, please design a strategy to sort people from 1 to n, such that the maximum earned money $c_{max} = \max_{1 \le i \le n} c_i$ is minimized. (Note: the 0-th person doesn't enroll in the sorting process, but a_0 always works for each c_i.)

   (a) Please design an algorithm based on greedy strategy to solve the above problem. (Write a pseudocode)

   (b) Prove your algorithm is optimal.

   Solution. Sort the people and make sure their $a_i b_i$ is in non-decreasing order.When two adjacent people have the same value of $a_i b_i$, then let the one with bigger b_i in the front.

   The i-th people will earn money $c_i = \frac{a_0 \times a_1 \times \cdots \times a_{i-1}}{b_i}$ and the $(i+1)$-th people will earn $c_{i+1} = \frac{a_0 \times a_1 \cdots \times a_i}{b_{i+1}}$.

   If swapping them ,the i-th people will earn $c'_i = \frac{a_0 \times a_1 \times \cdots \times a_{i-1}}{b_{i+1}}$ and the $(i+1)$-th people will earn $c'_{i+1} = \frac{a_0 \times a_1 \times \cdots \times a_{i-1} \times a_{i+1}}{b_i}$.

   If swapping them can make them earn less, we have $c_i > c'_i$, $c_{i+1} > c'_{i+1}$.

   we can get $b_{i+1} > b_i$ and $a_i b_i > a_{i+1} b_{i+1}$.

   Therefore people with smaller $a_i b_i$ or bigger b_i should be in the front.

   Algorithm 1: MinEarn($a[0, \cdots, n]$,$b[0, \cdots, n]$,n)

   Input: $a[1, \cdots, n]$,$b[1, \cdots, n]$,n
   Output: $a[1, \cdots, n]$,$b[1, \cdots, n]$,which make each person earn least.

   1 Sort the people to make their $a[i] \cdot b[i]$ in nondecreasing order.If several adjoining neighbors have the same $a[i] \cdot b[i]$, sort them to make their $b[i]$ in non-increasing order;
   2 **return** $a[1, \cdots, n], b[1, \cdots, n]$;

   □

2. **Interval Scheduling** is a classic problem solved by greedy algorithm and we have introduced it in the lecture: given n jobs and the j-th job starts at s_j and finishes at f_j. Two jobs are compatible if they do not overlap. The goal is to find maximum subset of mutually compatible jobs. Tim wants to solve it by sort the jobs in descending order of s_j. Is this attempt correct? Prove the correctness of such idea, or else provide a counter-example.

   Proof. Tim's idea is correct.

   Assumes that Tim's idea generate a set of compatible jobs A, which is not the best choice, and OPT is the best one.

   Assumes that A has m elements and OPT has n elements. Obviously $m \le n$.

Assumes that the last k elements in A and OPT are the same, and all the elements in front of $A[m-k+1]$ and $OPT[n-k+1]$ are different.

Tim's idea makes sure the start time of $A[m-k+1]$ is later than $OPT[n-k+1]$.Therefore it does not matter to change $OPT[n-k+1]$ into $A[m-k+1]$.

That is because $OPT[n-k]$ ends before $OPT[n-k+1]$ starts, therefore it ends earlier before $A[m-k+1]$ starts, too.

Therefore this change causes no harm to other jobs in OPT.

However, this time the last $k+1$ jobs in A and OPT are the same, which is contradictory to the assumption.

Therefore Tim's idea is the best.

□

3. There are n lectures numbered from 1 to n. Lecture i has duration (course length) t_i and will close on d_i-th day. That is, you could take lecture i **continuously** for t_i days and must finish before or on the d_i-th day. The goal is to find the maximal number of courses that can be taken. (Note: you will start learning at the 1-st day.)

   Please design an algorithm based on greedy strategy to solve it. You could use the data structrue learned on Data Structrue course. You need to write pseudo code and prove its correctness.

   Solution. First sort the lectures so that their deadlines are in non-decreasing order.

   Maintain a list A for the selected course, and initialize it to be empty.

   Then examine the unselected lectures one by one from the one with the earliest deadline.

   If the newly-examined lecture can be finished before its deadline without conflicting with already-chosen ones,then append it to A.

   If the newly-examined lecture can not be finished before its deadline without conflict, then find out the most time-consuming lecture in the already-chosen ones. If its t_i is larger than the newly-examined one, then omit it and append the newly-examined one to A.

   The proof is as follows.

   Assumes that A given by the method above is not the best one, and OPT is the best one instead.

   Omit the break time between two adjacent lectures in OPT, and we can get OPT'.

   The number of lectures in OPT' is not smaller than that of OPT', because by omitting the break time, lectures in OPT' end at least as early as their counterpart in OPT, if not earlier.

   Therefore OPT' is also the best choice.

   For the lectures that both OPT' and A choose, A sort them by their deadlines in nondecreasing order, which is better than OPT'.

   Because for any two lectures with durations t_1 and t_2, and deadlines d_1 and $d_2(d_1 < d_2)$, if choosing them in nondecreasing order, it requires that $t_1 + t_2 < d_2$. Otherwise it requires $t_1 + t_2 < d_1$, which is harder to satisfy. Therefore if these two lectures are both chosen, sort them by deadline in nondecreasing order is the best choice.

   For lectures chosen by A but not by OPT', A choose a lecture less time-consuming and with later deadline, which is not worse than OPT'.

   Therefore A is at least as good as OPT'.

Therefore A is the best choice.

Algorithm 2: FindMax($t[1,\cdots,n]$,$d[1,\cdots,n]$,n,A)

Input: $t[1,\cdots,n]$,$d[1,\cdots,n]$,n,A
Output: $count$ = maximal number of courses that can be taken.

```
1  count ← 0;
2  lecture[1,···,n];
3  A ← φ;
4  clock ← 0;
5  for i = 1 to n do
6      lecture[i].duration = t[i];
7      lecture[i].deadline = d[i];
8  Sort lecture[1,···,n] by deadline so that
     lecture[1].deadline ≤ lecture[2].deadline ≤ ··· ≤ lecture[n].deadline.
9  for j = 1 to n do
10     if lecture[j].deadline − lecture[j].duration ≥ clock then
11         count ← count + 1;
12         A[count] = lecture[j];
13         clock ← clock + A[count].duration;
14         rank ← count;
15         while rank ≠ 1 and A[rank].duration > A[⌊rank/2⌋].duration do
16             swap(A[rank], A[ ⌊rank/2⌋ ]);
17             rank ← ⌊rank/2⌋;
18     else
19         if lecture[j].duration < A[1].duration then
20             clock ← clock − A[1].duration + lecture[j].duration;
21             A[1] ← lecture[j];
22             rank ← 1;
23             while
                 A[rank].duration < A[rank × 2].duration or A[rank].duration <
                 A[rank × 2 + 1].duration do
24                 if A[rank × 2].duration > A[rank × 2 + 1].duration then
25                     swap(A[rank], A[rank × 2]);
26                     rank ← rank × 2;
27                 else
28                     swap(A[rank], A[rank × 2 + 1]);
29                     rank ← rank × 2 + 1;
30 return count;
```

4. Let $S_1, S_2, \ldots, S_n$ be a partition of S and $k_1, k_2, \ldots, k_n$ be positive integers. Let $\mathcal{I} = \{I : I \subseteq S, |I \cap S_i| \leq k_i \text{ for all } 1 \leq i \leq n\}$. Prove that $\mathcal{M} = (S, \mathcal{I})$ is a matroid.

Proof.

- **Statement.** A, B are two sets, then $A \subseteq B \Rightarrow |A| \leq |B|$

Proof:

Because $A \subseteq B$, therefore $B = A + (B - A)$. Therefore $|B| = |A| + |B - A|$.
Because $|B - A| \geq 0$, therefore $|B| \geq |A|$. Therefore $|A| \leq |B|$.

- For any $B \in \mathcal{I}$, $A \subseteq B$, we have

$$\forall x, x \in A \cap S_i \Leftrightarrow x \in A \wedge x \in S_i \Rightarrow x \in B \wedge x \in S_i \Leftrightarrow x \in B \cap S_i (i = 1, 2, \cdots, n)$$

Therefore $A \cap S_i \subseteq B \cap S_i$.
Therefore $|A \cap S_i| \leq |B \cap S_i|$.
Because $|B \cap S_i| \leq k_i$, then we have $|A \cap S_i| \leq k_i$, namely $A \in \mathcal{I}$.
Therefore $A \subseteq B, B \in \mathcal{I} \Rightarrow A \in \mathcal{I}$.
The heredity property is satisfied.

- Assume that there exist $A, B \in \mathcal{I}$, $|A| < |B|$, which satisfy that $\forall x \in B \setminus A, A \cup \{x\} \notin \mathcal{I}$.
Because $A \in S$, $B \in S$, S_i's $(1 \leq i \leq n)$ are a partition of S,
therefore
  - $\forall x \in S, \exists$ unique i_x, which satisfies $x \in S_{i_x}$.
  - $\forall x \in B \setminus A, \exists$ unique i_x, which satisfies $x \in S_{i_x}$
  - $|A| = \sum_{i=1}^{n} |A \cap S_i|$, $|B| = \sum_{i=1}^{n} |A \cap S_i|$.

  For all i which $1 \leq i \leq n$, it may have the following condition.
  - $\exists x_i \in S_i$, and $x_i \in B \setminus A$.
    Because x_i can not be selected as the x to make $A \cup \{x\} \in \mathcal{I}$, namely $|(A \cup \{x\}) \cap S_i| \leq k_i (1 \leq i \leq n)$,
    therefore $|A \cap S_i| = k_i$.
    Because $B \in \mathcal{I}$, therefore $|B \cap S_i| \leq k_i = |A \cap S_i|$.
  - $\exists x_i \in S_i$, but For any $x_i \in S_i$, $x_i \notin B \setminus A$.
    This time $|B \cap S_i| = |(B \cap A) \cap S_i| + |(B \setminus A) \cap S_i| = |(B \cap A) \cap S_i| + 0 \leq |A \cap S_i| \leq k_i$.
  - There does not exist any x, which $x \in S_i$.
    This time $|B \cap S_i| = 0 \leq |A \cap S_i|$.

  Therefore $|B| = \sum_{i=1}^{n} |B \cap S_i| \leq \sum_{i=1}^{n} |A \cap S_i| = |A|$.
  But according to the assumption,$|B| > |A|$.
  Therefore there must exist A, B which satisfy that $|A| < |B|$ and $\exists x \in B \setminus A, A \cup \{x\} \in \mathcal{I}$, which means the exchange property is satisfied.
  Therefore $(S, \mathcal{I})$ is a matroid.

□

Remark: You need to include your .pdf and .tex files in your uploaded .rar or .zip file.