Lab03-Matroid & DP

CS2308-Algorithm and Complexity, Xiaofeng Gao, Spring 2023.

∗ Contact TA Jiasen Li for any questions. Include both your .pdf and .tex files in the uploaded .rar or .zip file.

∗ Name:张祥东 521030910206 Time Spent on this Lab:5-6h

1. (Matroid) Provide an example of $(S, \mathbf{C})$, which is an independent system but not a matroid. Give an instance of S such that $v(S) \neq u(S)$ (should be different from the examples in class).

   Definition 1. ***S*** *consists of n objects,which have weight.*

   Definition 2. ***C*** *consists of subsets of S the weight summation of which is less than or equal to the given weight limit* ***X****.*

   For the subset B and the subset A of S. If $B \subset A$ $and A \in C$:

   Due to $A \in C$,the summation of elements in A $\leq$ the given X. And due to $B \subset A$,obviously the summation of elements in B $\leq$ the given X. Namely

   $$B \subset A \ and \ A \in C \rightarrow B \in C$$

   It's hereditary property,thus the system is a independent system.

   However,it's not a matroid.

   Let us give a concrete instance to illustrate this.

   - X = 10
   - n = 10 and weight$(a_1) = 5$,weight$(a_2) = 5$,$weight(a_3) = \cdots = weight(a_{10}) = 1$

   Easy to find that

   $$u(S) = 2$$
   $$v(S) = 8$$

   We know (S, C) is a matroid iff for any $F \subseteq S$,$u(F) = v(F)$.

   So this independent system is not a matroid.

2. (Knapsack & Greedy) Let $A = [a_1, a_2, \cdots, a_n]$, $weight(a_i)$, $volume(a_i) \in \mathbb{Z}^+$, $\forall 1 \leq i \leq n$. You should find a subset of m items with maximum density. Your time complexity should be at most $O(n \log n)$. Give your algorithm in the form of pseudo-code, prove the correctness and analyze its time complexity.

Algorithm 1: find the maximum desity subset of size m

Input: $A = [a_1, a_2, \ldots, a_n]$,weight,volume
Output: DensityNeeded and the solution

$maxDensity \leftarrow 0$;
$minDensity \leftarrow \infty$;
for $i \leftarrow 1$ *to* n **do**
    $maxDensity \leftarrow max(maxDensity, \frac{weight(a_i)}{volume(a_i)})$;
    $minDensity \leftarrow min(minDensity, \frac{weight(a_i)}{volume(a_i)})$;
$b \leftarrow$ new array of size n;
solution $\leftarrow$ new array of size m and it is not assigned now;
while $abs(maxDensity - minDensity) >= 1e-5$ **do**
    $mid \leftarrow (maxDensity + minDensity)/2$;
    for $i \leftarrow 1$ *to* n **do**
        $b_i \leftarrow (weight(a_i) - mid * volume(a_i), i)$;
    sort first element in the the array b by the relation $\geq$.
    $isValid \leftarrow 0$
    for $i \leftarrow 1$ *to* m **do**
        $isValid \leftarrow isValid + b_i.first$;
    if $isValid < 0$ **then**
        maxDensity = mid;
    else
        $solution \leftarrow b[1, \ldots, m].second$;
        minDensity = mid;
if *solution has not been assigned* **then**
    for $i \leftarrow 1$ *to* n **do**
        $b_i \leftarrow (weight(a_i) - minDensity * volume(a_i), i)$;
    sort first element in the the array b by the relation $\geq$.
    $solution \leftarrow b[1, \ldots, m].second$;
return minDensity,solution;

We take **density needed** denoting the maximum density that we need to figure out in this quesion.

Here comes the proof of the correctness of our algorithm:

Obviously,the maximum density that satisfiy the demand in the question should satisfiy

$$min\{\frac{weight(a_i)}{volume(a_i)}\} \leq density\ needed \leq max\{\frac{weight(a_i)}{volume(a_i)}\}$$

.

For every density that is less than the *density needed*,it can be satisfied,and for every density that is greater than the *densityneeded*,it cannot be satisfied.Thus we can do binary search to find the answer.

Only if we find the maximum density that can be satisfied by the array a,the density needed must be that number.So to prove our algorithm's correctness,we've already known through previous state that our algorithm can give the right answer if the operation in the algorithm is right.

So next,we prove the correctness of our operations.

For mid,we want to test

$$\begin{aligned}
&mid\ is\ a\ proper\ answer\\
\iff\ &there\ exist\ a\ sequence\ a_{i_1}, ai_2, \ldots, a_{i_m} that \frac{\sum_{j=1}^{m} weight(a_{i_j})}{\sum_{j=1}^{m} volume(a_{i_j})} \geq mid\\
\iff\ &there\ exist\ a\ sequence\ a_{i_1}, ai_2, \ldots, a_{i_m} that \sum_{j=1}^{m} weight(a_{i_j}) \geq mid * \sum_{j=1}^{m} volume(a_{i_j})\\
\iff\ &there\ exist\ a\ sequence\ a_{i_1}, ai_2, \ldots, a_{i_m} that \sum_{j=1}^{m} (weight(a_{i_j}) - mid * volume(a_{i_j})) \geq 0
\end{aligned} \tag{1}$$

So if we could find out whether there exist a sequence of $a_{i_1}, ai_2, \ldots, a_{i_m}$ that statisfies $\sum_{j=1}^{m}(weight(a_{i_j}) - mid * volume(a_{i_j})) \geq 0$,the legality of the mid can be proved.
So we get new array b for every mid

$$b[\cdot] \leftarrow weight(a[\cdot]) - mid * volume(a[\cdot])$$

we try to find the biggest summation subset of b with size m using greedy.By sort the array b with $\geq$ and calculate the first m elements' summation.If the summation is smaller than 0,the mid is illegal,or it's legal.

As for why the biggest summation subset can be found with greedy,we'd like to prove it by contradiction.

First,we get sequences of size m. The optimal sequence taken as a^{opt}.The sequence obtained by greedy taken as a^{alg}. Sort the a^{opt} and a^{alg} by the relation geq.

We assume the $a^{opt} \neq a^{alg}$

We assume that $a_i^{opt} = a_i^{alg}$ until i,namely

$$a_i^{opt} \neq a_i^{alg}$$

And from our selection method,the inequality $a_i^{alg} > a_i^{opt}$ must exists.So if we replace the a_i^{opt} with a_i^{alg},the optimal solution will be better.It's impossible for a optimal solution to get better.

So the i doesn't exist,the assumption is wrong,namely $a^{opt} = a^{alg}$.

Above all,the proof of my algorithm's correctness is completed.

As for the time complexity.By limit the precision into $1e-5$,we can consider the complexity for binary search is $O(k * 1e5) = O(1)$ times.

*k is the maxDensity - minDensity in my pseudo-code and it can be constrained by a constant here,namely $k = O(1)$.

In the While part,the for in it is $O(n+m)$ and the sort in it is $O(n \log n)$.

So the time complexity is

$$O(k * 1e5 * (n + m + n \log n)) = O(n \log n)$$

It's

$$O(n \log n)$$

.

3. (DP Application) Assuming you are a programmer for Word, you are now interested in studying how to make document formatting neatly. We model the problem as follows: Denote the article as a sequence $A = \langle s_1, s_2, \cdots, s_n \rangle$ of strings, each with a given length satisfying $len(s_i) \leq k$, $\forall 1 \leq i \leq n$. Your job is typesetting these strings into a Word file and deciding how many words are in each line. There should be a space between every two adjacent words in a line. Commonly, a line looks good with m characters, including spaces. During the composition, if the number of characters in a line is m', there is a penalty of $(m - m')^2$. Design an algorithm to minimize the sum of penalties with time complexity at most $O(n \log n)$. Give your algorithm in the form of pseudo-code and analyze its time complexity.

Algorithm 2: DP application

Input: a sequence A = $< s_1, s_2, \ldots, s_n >$ of strings and the given m

Output: Solution to determine after which string the line change should be applied to get the minimize penalty and the minimum penalty

$pre \leftarrow size\ n\ +\ 1$ and index from 0 to n with initial value 0;
for $i \leftarrow 1\ to\ n$ **do**
    $pre_i \leftarrow pre_{i-1} + 1$;
$from \leftarrow$ size n and with initial value -1;
$dp \leftarrow size\ n+\ 1$ and index from 0 to n with initial value 0;
$dp_1 \leftarrow dp_0 + (m - (pre_1 - 1))^2$
$convexPack$ is a convex pack and maintain this convexPack in $O(\log n)$ for every added point operation using a balance tree.
$convexPack \leftarrow point(pre_0, dp_0 + (pre_0 - 1)^2 + 2mpre_0 + 2m + m^2)$;
$convexPack \leftarrow point(pre_1, dp_1 + (pre_1 - 1)^2 + 2mpre_1 + 2m + m^2)$;
$from[1] \leftarrow 0$;
$pointStack$ is a point sequence maintaining counterclockwise point sequence in the convexPack's edge and renewed danamically with the convexpack **which only contains the points laying at the down convex pack**.
for $i \leftarrow 2\ to\ n$ **do**
    $k \leftarrow 2pre_i$;
    $l \leftarrow 2; r \leftarrow sizeof(pointStack)$;
    while $l < r$ **do**
        $mid \leftarrow (l + r)/2$;
        $cur = (pointStack_{mid}.second - pointStack_{mid-1}.second)/(pointStack_{mid}.first - pointStack_{mid-1}.first)$;
        if $cur < k$ **then**
            $l \leftarrow mid + 1$;
        else
            $r \leftarrow mid$;
    $ans1 \leftarrow -k * pointStack_l.first + pointStack_l.second$;
    $ans2 \leftarrow -k * pointStack_{l-1}.first + pointStack_{l-1}.second$;
    if $ans1 < ans2$ **then**
        $dp_i = ans1 + pre_i^2 - (2m + 2)pre_i$;
        from[i] $\leftarrow$ l;
    else
        $dp_i = ans2 + pre_i^2 - (2m + 2)pre_i$;
        from[i] $\leftarrow$ l - 1;
    $convexPack \leftarrow add\ point(pre_i, dp_i + (pre_i - 1)^2 + 2mpre_i + 2m + m^2)$;
solution $\leftarrow$ empty;
$cur \leftarrow from[n]$;
while $from[cur] \neq 0$ **do**
    solution.push_front(cur);
    cur $\leftarrow$ from[cur];
return dp_n,solution;

Definition 3. ***pre*** $\rightarrow$ *an array of size n + 1.*pre_i *denotes the characters before* i_{th} *string (including* i_{th} *string) and add i - 1 characters which are spaces among i strings.*

Definition 4. ***dp*** $\rightarrow$ *an array of size n + 1.*dp_i *denotes the minimize penalty when take the* i_{th} *string as the end string.*

We consider this question in one demesion perspective,here comes the OPT function

$$dp_i = \begin{cases} min_{j=0}^{i-1}\{dp_j + (m - (pre_i - pre_j - 1))^2\} & i >= 1 \\ 0 & i = 0 \end{cases}$$

Using this OPT function,we can solve this question in $O(n^2)$.

To accerlerate this process,we can reconsider the following equation

$$dp_i = min_{j=0}^{i-1}\{dp_j + (m - (pre_i - pre_j - 1))^2\}$$

$$dp_i \geq dp_j + (m - (pre_i - pre_j - 1))^2$$

$$dp_i + 2mpre_i - pre_i^2 - 2pre_i \geq dp_j + (pre_j - 1)^2 + 2mpre_j + m^2 + 2m - 2pre_i pre_j$$

Record the above inequality as the following form:

$$tar(i) \geq h(j) + f(i)g(j)$$

$$tar(i) = dp_i + 2mpre_i - pre_i^2 - 2pre_i$$

$$h(j) = dp_j + (pre_j - 1)^2 + 2mpre_j + m^2 + 2m$$

$$f(i) = -2pre_i$$

$$g(j) = pre_j$$

Then tar(i) equals to the $max_{j=0}^{i-1}\{h(j) + f(i)g(j)\}$,we take k = h(j) + f(i)g(j),namely

$$h(j) = -f(i)g(j) + k$$

After find the tar(i),through

$$dp_i = tar(i) - 2mpre_i + pre_i^2 + 2pre_i$$

we can easily get the dp_i.

So we only need to record the former point(g(j),h(j)) and using the slope of value -f(i) to find the point whose intercept is the minimum.For i,we've maintained the previous i - 1 point and find a convex pack for this points,to find the best point we only need to binary the points in the convex pack and using the -f(i) as the search basis.Thus we can update dp_i in $O(\log n)$. **It's noteworthy that the** $-f(i) = 2pre_i$ **which is positive,so in our pseudo-code,we only need to main the points laying at the down part of convex pack.**

Above all,we maintained our convex pack in $O(n \log n)$ and do n - 1 times binary search with $O(\log n)$ each,so the time complexity is

$$O(n \log n + (n-1) \log n)$$

It's

$$O(n \log n)$$

4. (DP & Binary Search) Let $A = [a_1, a_2, \cdots, a_n]$, $a_i \in \mathbb{Z}$. A sub-sequence of A can be described as $A' = [a_{i_1}, a_{i_2}, \cdots, a_{i_k}]$, where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. An increasing sub-sequence of A satisfies $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$. Find the longest increasing sub-sequence in worst case $O(n \log n)$ time complexity. Give your algorithm in the form of pseudo-code and analyze its time complexity.

Algorithm 3: LIS problem

Input: A$[a_1, a_2, \ldots, a_n]$
Output: The longest increasing sub-sequence in A$[\cdot]$

```
dp[1, 2, ..., n] ← {∞, −1} foreach in dp[·];
pre[1, 2, ..., n] ← -1 foreach in pre[·];
for i ← 1 to n do
    l ← 1; r ← n;
    while l < r do
        m ← (l+r)/2;
        if dp_m.first < a_i then
            l ← m + 1;
        else
            r ← m
    dp_l ← {a_i, i};
    pre_i ← dp_{l−1}.second;
ans ← 1;
while dp_ans.first ≠ ∞ do
    ans ← ans + 1;
ans ← ans − 1;
solution ← empty deque;
cur ← dp_ans.second;
while cur ≠ −1 do
    solution.push_front(a_cur);
    cur ← pre_cur;
return solution;
```

Here comes the OPT relationship

$$OPT[i] = \begin{cases} max_{...}\{OPT[j] + 1\} & i \neq 1 \ or \ 0 \\ 1 & i = 1 \\ 0 & i = 0 \end{cases}$$

***j in 1 to i - 1 for every j that a[j] < a[i].If j doesn't exists,j = 0.**

If we do this by enumerating,there will be $O(n)$ iterations and in each iteration,the enumerating times will be $O(n)$.If so,the time complexity will be $O(n^2)$.

But by using binary search to accelerate the times of enumerate operation,we get $O(\log n)$ operation in each iteration.

As for why the binary search can be used,we know that the optimal only have something to do with the sequence that is longest and its last element is smaller than the current one. So it can be used.

The overall the time complexity is

$$O(n \log n)$$