Lab04-Matroid

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

∗ If there is any problem, please contact TA Yiming Liu.

∗ Name:Hongjie Fang Student ID:518030910150 Email: galaxies@sjtu.edu.cn

1. Give a directed graph $G = (V, E)$ whose edges have integer weights. Let $w(e)$ be the weight of edge $e \in E$. We are also given a constraint $f(u) \geq 0$ on the out-degree of each node $u \in V$. Our goal is to find a subset of edges with maximal weight, whose out-degree at any node is no greater than the constraint.
   (a) Please define independent sets and prove that they form a matroid.
   (b) Write an optimal greedy algorithm based on Greedy-MAX in the form of *pseudo code*.
   (c) Analyze the time complexity of your algorithm.

Solution. Here are the answers to the questions.

(a) First we construct a edge set E' based on edge set E, but we abandon the negative-weighted edges, that is,

$$E' = \{e \mid e \in E, w(e) \geq 0\}$$

Then we define independent subsets and the collection of independent subsets as follows.

Definition 1. *A subset $I \subseteq E'$ is independent if and only if for every node $u \in V$, the out-degree at node u in edge set I does not exceed the constraint $f(u)$, that is,*

$$|\{(u, v) \mid v \in V, (u, v) \in I\}| \leq f(u) \qquad (\forall u \in V)$$

And we define $\mathbf{C}$ as the collection of all independent subsets, that is,

$$\mathbf{C} = \{I \mid I \subseteq E', I \text{ is independent}\}$$

We also define M as an independent system constructed by edge set E' and $\mathbf{C}$, that is,

$$M = (E', \mathbf{C})$$

Lemma 1 (Hereditary)**.** *Collection $\mathbf{C}$ is hereditary.*

Proof. $\forall B \in \mathbf{C}$, we know that B is an independent subset, that is, for every node $u \in V$, the out-degree at node u in edge set B does not exceed the constraint $f(u)$. $\forall A \subseteq B$, for every node $u \in V$, the out-degree at node u in edge set A is no more than that in edge set B, so it does not exceed the constraint $f(u)$. Therefore, $A \in \mathbf{C}$, which completes the proof of hereditary. □

Lemma 2 (Exchange Property)**.** *The independent system $M = (E', \mathbf{C})$ satisfies the exchange property.*

Proof. $\forall A, B \in \mathbf{C}$, suppose that $|A| < |B|$. There must exist at least one node u satisfying the out-degree at node u in edge set B is **strictly more than** that in edge set A, otherwise we will derive $|B| \leq |A|$, which contradicts the premise. Therefore, there exists at least one edge e started from u satisfying $e \in B$ but $e \notin A$. Then $A \cup \{e\} \in \mathbf{C}$ because in node u, we have

$$\begin{aligned}|\{(u, v) \mid v \in V, (u, v) \in A \cup \{e\}\}| &\leq |\{(u, v) \mid v \in V, (u, v) \in A\}| + 1 \\ &\leq |\{(u, v) \mid v \in V, (u, v) \in B\}| \\ &\leq f(u)\end{aligned}$$

and in all other nodes u', we have

$$|\{(u',v) \mid v \in V, (u',v) \in A \cup \{e\}\}| \le |\{(u',v) \mid v \in V, (u',v) \in A\}|$$
$$\le f(u') \qquad (\forall u' \in V, u' \ne u)$$

Therefore, there exists $e \in B \backslash A$ such that $A \cup \{x\} \in \mathbf{C}$. The property is proved. $\square$

Theorem 3. *The independent system $M = (E', \mathbf{C})$ is actually a matroid.*

Proof. We have proved the hereditary of collection $\mathbf{C}$ in Lemma 1 and the exchange property of M in Lemma 2, therefore, $M = (E', \mathbf{C})$ is a matroid. $\square$

(b) We define the answer as an edge set including all the edges we selected.

Lemma 4. *The optimal answer S^* only includes edges in E'.*

Proof. If an optimal answer S^* includes an edge $e_0 \in E \backslash E'$, then we have $w(e_0) < 0$. Then we constructed another answer $S^* \backslash \{e_0\}$, and we have

$$\sum_{e \in S^*} w(e) = w(e_0) + \sum_{e \in S^* \backslash \{e_0\}} w(e) < \sum_{e \in S^* \backslash \{e_0\}} w(e)$$

Therefore, the answer $S^* \backslash \{e_0\}$ is better than the optimal answer S^*, which causes a contradiction. Therefore, the optimal answer only includes edges in E'. $\square$

Up to now, we have proved that the optimal answer only selected edges from E'. So we can only consider about edge set E', rather than the full edge set E. Therefore, we can write an Greedy-MAX algorithm as follows (Alg. 1) to solve the question.

Algorithm 1: Greedy-MAX

Input: The Graph $G = (V, E)$ and the constraints $f(u)$ of every node u
Output: A subset of edges with maximal weight, whose out-degree at any node is no greater than the constraint

```
Abandon all the negative-weighted edge in E and form a new edge set E'.
Sort all elements in E' into ordering w(e_1) ≥ w(e_2) ≥ ... ≥ w(e_m).
S ← ∅;
for i = 1 to m do
    if S ∪ {e_i} ∈ C then
        S ← S ∪ {e_i}
return S;
```

We have proved that $M = (E', \mathbf{C})$ is a matroid, so according to the *Greedy Theorem for Independent System*, we know that our Greedy-MAX algorithm provides us an optimal answer. Therefore, the Greedy-MAX algorithm (Alg. 1) is **correct**.

(c) With a few optimizations, the algorithm has a time complexity of $O(|E| \log |E|)$.

- The step of abandoning negative-weighted edges takes $O(|E|)$ time.
- The step of sorting edges in E' takes $O(|E| \log |E|)$ time at most.
- Without any optimization, the step of choosing edge takes $O(|V|)$ time for every edge, so the whole step takes $O(|V||E|)$ time. But if we **record the current out-degree of every node in an array**, then for every edge we only need $O(1)$ time to check if the new set belongs to $\mathbf{C}$, therefore we only need $O(|E|)$ time in this step.

$\square$

2. Let X, Y, Z be three sets. We say two triples (x_1, y_1, z_1) and (x_2, y_2, z_2) in $X \times Y \times Z$ are *disjoint* if $x_1 \neq x_2$, $y_1 \neq y_2$, and $z_1 \neq z_2$. Consider the following problem:

   Definition 2 (MAX-3DM)**.** *Given three disjoint sets X, Y, Z and a nonnegative weight function $c(\cdot)$ on all triples in $X \times Y \times Z$,* ***Maximum 3-Dimensional Matching*** *(MAX-3DM) is to find a collection $\mathcal{F}$ of disjoint triples with maximum total weight.*

   (a) Let $D = X \times Y \times Z$. Define independent sets for MAX-3DM.

   (b) Write a greedy algorithm based on Greedy-MAX in the form of *pseudo code.*

   (c) Give a counterexample to show that your Greedy-MAX algorithm in Q. 2b is not optimal.

   (d) Show that: $\max_{F \subseteq D} \frac{v(F)}{u(F)} \leq 3$. (Hint: you may need Theorem 5 for this subquestion.)

   Theorem 5. *Suppose an independent system $(E, \mathcal{I})$ is the intersection of k matroids $(E, \mathcal{I}_i)$, $1 \leq i \leq k$; that is, $\mathcal{I} = \bigcap_{i=1}^{k} \mathcal{I}_i$. Then $\max_{F \subseteq E} \frac{v(F)}{u(F)} \leq k$, where $v(F)$ is the maximum size of independent subset in F and $u(F)$ is the minimum size of maximal independent subset in F.*

   Solution. Here are the answers to the questions.

   (a) We define the independent subsets of MAX-3DM as follows.

   Definition 3. *A subset $I \subseteq D$ is independent if and only if every two triples in I are disjoint.*

   (b) We define $\mathbf{C}$ as the collection of all independent subsets, and define M as $(D, \mathbf{C})$. Therefore M is a independent system since $\mathbf{C}$ is hereditary obviously. Then we can write a Greedy-MAX algorithm (Alg. 2) to solve the problem.

   Algorithm 2: Greedy-MAX

   Input: Three disjoint set X, Y, Z and a nonnegative weight function $c(\cdot)$ on all triples in $D = X \times Y \times Z$

   Output: A collection $\mathcal{F}$ of disjoint triples with maximum total weight

   Sort all the triples in D into ordering $c(d_1) \geq c(d_2) \geq ... \geq c(d_m)$.
   $\mathcal{F} \leftarrow \varnothing$;
   for $i = 1$ ***to*** m **do**
       if $\mathcal{F} \cup \{d_i\} \in \mathbf{C}$ **then**
           $\mathcal{F} \leftarrow \mathcal{F} \cup \{d_i\}$
   return $\mathcal{F}$;

   Actually, the answer that Greedy-MAX algorithm (Alg. 2) provides us is not optimal sometimes (i.e., it is **incorrect**). We will discuss it further in answers to the following questions.

   (c) Here is a counter-example.

   Suppose $X = \{1, 2\}, Y = \{3, 4\}, Z = \{5, 6\}$, and we define $c(\cdot)$ as follows.

$$
\begin{aligned}
&c((1,3,5)) = 8, \quad c((1,3,6)) = 9, \quad c((1,4,5)) = 0, \quad c((1,4,6)) = 0 \\
&c((2,3,5)) = 0, \quad c((2,3,6)) = 0, \quad c((2,4,5)) = 0, \quad c((2,4,6)) = 7
\end{aligned}
$$

   In Greedy-MAX algorithm, we will have the answer $\mathcal{F} = \{(1,3,6), (2,4,5)\}$ and the total weight of $\mathcal{F}$ is 9. But the optimal answer is $\mathcal{F}^* = \{(1,3,5), (2,4,6)\}$ and the total weight of $\mathcal{F}^*$ is $8 + 7 = 15$.

(d) We first define i-independent ($i = 1, 2, 3$) as follows.

Definition 4 (i-independent)**.** *A subset $I \subseteq D$ is i-independent ($i = 1, 2, 3$) if and only if the numbers in the i-th dimension of every two triples in I are different.*

For instance, $I_{e1} = \{(1, 2, 3), (1, 2, 4)\}$ is 3-independent, and $I_{e2} = \{(2, 3, 4), (1, 5, 4)\}$ is 1-independent and 2-independent.

We define $\mathbf{C}_i$ ($i = 1, 2, 3$) as the collection of all i-independent subset, and M_i as an independent system constructed by D and $\mathbf{C}_i$, that is, $M_i = (D, \mathbf{C}_i)$ ($i = 1, 2, 3$).

Lemma 6 (Hereditary)**.** *Collection $\mathbf{C}_i$ is hereditary for $i = 1, 2, 3$.*

Proof. Given an i from $\{1, 2, 3\}$. $\forall B \in \mathbf{C}_i$, we know that B is an i-independent subset, that is, the numbers in i-th dimension of every two triples in B are different. $\forall A \subseteq B$, the numbers in i-th dimension of every two triples in A are still different. Therefore, $A \in \mathbf{C}_i$, which completes the proof of hereditary. $\square$

Lemma 7 (Exchange Property)**.** *The independent system $M_i = (D, \mathbf{C}_i)$ satisfies the exchange property for $i = 1, 2, 3$.*

Proof. Given an i from $\{1, 2, 3\}$. $\forall A, B \in \mathbf{C}_i$ and suppose that $|A| < |B|$. Suppose $\dim_j(S)$ means the set of numbers in j-th dimension of S, that is,

$$\dim_j(S) = \{a_j \mid (a_1, a_2, ..., a_j, ...) \in S\}$$

There must exist at least one element $c \in \dim_i(D)$ satisfying $c \in \dim_i(B)$ but $c \notin \dim_i(A)$, otherwise we will derive that $|B| \leq |A|$, which contradicts the premise. Therefore, there exists an element $d \in B$ with its i-th dimension being c, and we have $d \notin A$ because $c \notin \dim_i(A)$. Then $A \cup \{d\} \in \mathbf{C}_i$ because the number in i-th dimension of d (that is c) is different from the number of i-th dimension of any element in A (the set of all such numbers is $\dim_i(A)$).

Therefore, there exists $d \in B \backslash A$ such that $A \cup \{d\} \in \mathbf{C}_i$. The property is proved. $\square$

Theorem 8. *The independent system $M_i = (D, \mathbf{C}_i)$ is actually a matroid, for $i = 1, 2, 3$.*

Proof. Given an i from $\{1, 2, 3\}$. We have proved the hereditary of collection $\mathbf{C}_i$ in Lemma 6 and the exchange property of M_i in Lemma 7, therefore, $M_i = (D, \mathbf{C}_i)$ is a matroid. $\square$

An obvious fact is that $\cap_{i \in \{1,2,3\}} \mathbf{C}_i = \mathbf{C}$, because if the numbers of every dimension in every two triples in a set are different, then every two triples in a set are disjoint, so the set is independent.

So the independent system M is the intersection of 3 matroids M_1, M_2, M_3. Then according to Theorem 5, we know that

$$\max_{F \subseteq D} \frac{v(F)}{u(F)} \leq 3$$

$\square$

3. **Crowdsourcing** is the process of obtaining needed services, ideas, or content by soliciting contributions from a large group of people, especially an online community. Suppose you want to form a team to complete a crowdsourcing task, and there are n individuals to choose from. Each person p_i can contribute v_i ($v_i > 0$) to the team, but he/she can only work with up to c_i other people. Now it is up to you to choose a certain group of people and maximize their total contributions ($\sum_i v_i$).

   (a) Given $\mathrm{OPT}(i, b, c)$ = maximum contributions when choosing from $\{p_1, p_2, \cdots, p_i\}$ with b persons from $\{p_{i+1}, p_{i+2}, \cdots, p_n\}$ already on board and at most c seats left before any of the existing team members gets uncomfortable. Describe the optimal substructure as we did in class and write a recurrence for $\mathrm{OPT}(i, b, c)$.

   (b) Design an algorithm to form your team using dynamic programming, in the form of *pseudo code*.

   (c) Analyze the time and space complexities of your design.

Solution. Here are the answers to the questions.

(a) **Optimal Substructure**

- Case 1: **OPT selects person p_i.**
  - Collect contribution v_i;
  - Only happens when current situation does not make person p_i uncomfortable, that is, $b \leq c_i$;
  - Only happens when there is at least one seat left, that is, $c \geq 1$.
  - After selecting person p_i, the current optimal solution must include optimal solution to problem consisting of $(b+1)$ person from $p_i, p_{i+1}, \cdots, p_n$ on board and $\min(c-1, c_i - b)$ rest seats.
- Case 2: **OPT does not select person p_i.**
  - No contribution;
  - The current optimal solution must include optimal solution to problem consisting of b person from $p_i, p_{i+1}, \cdots, p_n$ on board and c rest seats.

Therefore, the **recurrence** is as follows (Eqn. (1)).

$$OPT(i,b,c) = \begin{cases} \max(OPT(i-1,b,c), \\ \qquad OPT(i-1, b+1, \min(c-1, c_i - b)) + v_i), & (i \geq 1, c \geq 1, b \leq c_i) \\ OPT(i-1, b, c), & (i \geq 1, b > c_i) \\ 0, & (i = 0) \end{cases} \tag{1}$$

(b) We provide the pseudo-codes of the algorithm to compute the maximum total contributions and find the solution. (Alg. 3, Alg. 4 and Alg. 5).

Algorithm 3: CrowdSourcing - Memorization

Input: n; $v_1, v_2, \cdots, v_n$; $c_1, c_2, \cdots, c_n$;
Output: Maximum total contributions and the solution

```
M[0, ·, ·] ← 0;
Contribution ← CrowdSourcing(n, 0, n);
Solution ← FindSolution(n, 0, n);
return Contribution, Solution;
```

Algorithm 4: CrowdSourcing(i, b, c)

```
if M[i, b, c] is uninitialized then
    M[i, b, c] ← CrowdSourcing(i − 1, b, c);
    if c ≥ 1 and b ≤ c_i then
        M[i, b, c] ← max(M[i, b, c], CrowdSourcing(i − 1, b + 1, min(c − 1, c_i − b)) + v_i);
return M[i, b, c];
```

Algorithm 5: FindSolution(i, b, c)

```
if i = 0 then
    return ∅;
if c ≥ 1 and b ≤ c_i then
    if M[i, b, c] < M[i − 1, b + 1, min(c − 1, c_i − b)] + v_i) then
        return FindSolution(i − 1, b + 1, min(c − 1, c_i − b)) ∪ {p_i};
    else
        return FindSolution(i − 1, b, c);
return FindSolution(i − 1, b, c);
```

(c) **Space Complexity** We only use an array of length n for solution recording and a $n \times n \times n$ array for memorization, so the overall space complexity is $O(n^3)$.

Time Complexity

- We have a total number $n \cdot n \cdot n = n^3$ of different states. Notice that we compute every state only once owing to memorization, that is, we invoke CrowdSourcing$(\cdot, \cdot, \cdot)$ at most n^3 times. Each invocation takes $O(1)$ time. Therefore, the total time complexity of CrowdSourcing$(\cdot, \cdot, \cdot)$ is $O(n^3)$.
- Notice that we only invoke FindSolution$(\cdot, \cdot, \cdot)$ exactly n times, because the first argument is decreasing by 1 every time and its original value is n. Each invocation takes $O(1)$ time. Therefore, the total time complexity of FindSolution$(\cdot, \cdot, \cdot)$ is $O(n)$.
- **Total:** The total time complexity of the algorithm is $O(n^3)$.

□

Here we provide better solutions to the Problem 3.

Solution. Notice that the number of persons in a team is determined by **the minimum c_i among the team members**.

We sort persons according to their c_i in a non-ascending order in the beginning and re-index them according to the sorting result, then the number of persons in a team is determined by **the rightmost person we choose**.

Then let's enumerate the rightmost person we choose, suppose it is person i. Then the answer is simple - to find at most c_i persons in person $1, 2, \cdots, (i-1)$ with the largest contribution value. We have learned Greedy algorithm, so it can be easily solved by sorting the rest person according to their v_i in a non-ascending order, and choosing the first c_i persons.

Let's find out the current time complexity. The sorting in the beginning need $O(n \log n)$ time. Each situation in enumerating need $O(n \log n)$ time because of the greedy algorithm, and we enumerate n times, so the overall time complexity of enumerating is $O(n^2 \log n)$. The total time complexity is $O(n^2 \log n)$.

We provide the pseudo-code (Alg. 6) to compute the maximum total contributions. The method of finding the solution is simple and we will not discuss it here.

Algorithm 6: CrowdSourcing - Improved

Input: n; $v_1, v_2, \cdots, v_n$; $c_1, c_2, \cdots, c_n$;
Output: Maximum total contributions

Sort persons according to their c_i in a not-ascending order, and re-index them according to the sorting result;
$Contribution \leftarrow 0$;
for $i = n$ ***downto*** 1 **do**
  Copy the array $\{v_1, v_2, \cdots, v_{i-1}\}$ into $\{v'_1, v'_2, \cdots, v'_{i-1}\}$;
  Sort $\{v'_1, v'_2, \cdots, v'_{i-1}\}$ into non-ascending order;
  $cnt \leftarrow v_i$;
  for $j = 1$ ***to*** $\min(c_i, i-1)$ **do**
    $cnt \leftarrow cnt + v'_j$;
  if $cnt > Contribution$ **then**
    $Contribution \leftarrow cnt$;
return $Contribution$;

This solution seems better than the previous one, but - we can optimize it to reach the time complexity of only $O(n \log n)$!

Let's find out what will be changed if we change our rightmost person from i to $(i-1)$.

- We cannot choose person i, and we must choose person $(i-1)$;
- We will choose c_{i-1} persons among person $1, 2, ..., (i-2)$, instead of choosing c_i persons among person $1, 2, ..., (i-1)$.

But - the most important thing - our choosing strategy won't change! It's always choose the person with the highest contribution first!

Then we need a data structure supporting:

- Insert a value v into the data structure;
- Delete a value v from the data structure;
- Find out the sum among the biggest value, the second biggest value, ..., the k-th biggest value in the data structure.

The balanced tree, such as AVL Tree, Red-Black Tree, works! They can support all these operation in $O(\log n)$ time once.

Specific Explanation: We will store the values according to themselves in the tree, and we record the size of every sub-tree and the sum of every sub-tree. It's easy to maintain these things when inserting or deleting in $O(\log n)$ time. When requiring the answer, we only need to find the rightmost k values in the tree. We will start from root and check if the right sub-tree has enough values, if true, then go to the right sub-tree and use the same strategies recursively. If false, then we must choose all the values in the right sub-tree, the sum of which is in the sum tag, and we go to the left sub-tree to select the rest values using the same strategies recursively. This process will only take $O(\log n)$ time.

The whole procedure of the algorithm is:

- Sort the persons according to their c_i in a non-ascending order.
- Insert their v_i into a balanced tree.
- Enumerate the rightmost person i we choose in an descending order.
  - Delete v_i from balanced tree.
  - Find at most c_i values with the biggest summation from the balanced tree.
  - Calculate the total contributions.
  - Check if it can update the current maximum total contributions.
- Then, we find out the maximum total contributions.

The pseudo-code of the algorithm is as follows (Alg. 7).

Algorithm 7: CrowdSourcing - Improved Again

Input: n; $v_1, v_2, \cdots, v_n$; $c_1, c_2, \cdots, c_n$;
Output: Maximum total contributions

Sort persons according to their c_i in a not-ascending order, and re-index them according to the sorting result;
$Contribution \leftarrow 0$;
$T \leftarrow \varnothing$;
`// ` T ` is a balanced tree.`
for $i = 1$ ***to*** n **do**
    Insert(T, v_i);
    `// Insert(`T`, `v`) means inserting ` v_i ` into tree ` T`.`
for $i = n$ ***downto*** 1 **do**
    Delete(T, v_i);
    `// Delete(`T`, `v`) means deleting ` v_i ` from tree ` T`.`
    $cnt \leftarrow v_i + \text{FindK}(T, \min(c_i, i-1))$;
    `// FindK(`T`, `k`) means finding the largest ` k ` elements in tree ` T`.`
    if $cnt > Contribution$ **then**
        $Contribution \leftarrow cnt$;
return $Contribution$;

The time complexity of the final algorithm (Alg. 7) is only $O(n \log n)$, which is a lot faster than $O(n^3)$ in our original algorithm (Alg. 3). What's more, its space complexity is only $O(n)$! □

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