Lab10-Approximation & Randomized Algorithm

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2019.

∗ If there is any problem, please contact TA Mingran Peng.
∗ Name:Tianyao Shi Student ID:517021910623 Email: sthowling@sjtu.edu.cn

1. Given a CNF Φ with n boolean variables $\{x_i\}_{i=1}^n$ and m clauses, with each clause consisting of 3 boolean variables. For example $\Phi = C_1 \wedge C_2 = (x_1 \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. Assume that Φ is satisfiable, the goal is to find the feasible assignment of $\{x_i\}_{i=1}^n$ with **fewest true boolean variables**.
   (a) Please formulate it into integer programming.
   (b) Design an approximation algorithm based on deterministing rounding. Choose its approximation ratio and explain. Pseudo code is needed.

Solution. (a) Let $\mathbf{C}$ denote the set of all clauses. For each clause $c \in \mathbf{C}$, let S_c^+ and S_c^- denote the set of boolean variables occurring nonnegated and negated in c accordingly. The truth assignment is encoded by $\mathbf{y}$. Picking $y_i = 1$ denotes setting x_i to be True, and $y_i = 0$ denotes setting x_i to be False. Then we can formulate an integer linear programming as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^n y_i \\ \text{subject to} \quad & \forall c \in \mathbf{C} : \sum_{x_i \in S_c^+} y_i + \sum_{x_i \in S_c^-} (1 - y_i) \geq 1 \\ & \forall i : y_i \in \{0, 1\} \end{aligned}$$

(b) We can relax the ILP to be an LP as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^n y_i \\ \text{subject to} \quad & \forall c \in \mathbf{C} : \sum_{x_i \in S_c^+} y_i + \sum_{x_i \in S_c^-} (1 - y_i) \geq 1 \\ & \forall i : 0 \leq y_i \leq 1 \end{aligned}$$

Note that the constraint $y_i \leq 1$ is actually redundant. Proof of this claim is that, suppose there is a solution where some $y_i > 1$. In this case we have that $1 - y_i < 0$. Then why not set it to exactly 1? If $x_i \in S_c^+$, reducing y_i to 1 still ensures the sum of all y_i and $1 - y_i$ to be greater than or equal to 1. If $x_i \in S_c^-$, then reducing y_i to 1 turns $1 - y_i$ from a negative number to be 0, which makes the sum of all y_i and $1 - y_i$ to be even greater. In both case, the constraint will be satisfied, yet we derive a smaller $\sum_{i=1}^n y_i$. Therefore the nature of this problem restricts y_i to be smaller than or equal to 1.

Then there follows the deterministic rounding algorithm. See Algorithm 1.

Algorithm 1: Deterministic LP-rounding algorithm

```
Input: A CNF $\Phi$ with $n$ boolean variables $\{x_i\}_{i=1}^n$ and $m$ clauses
       $\mathbf{C} = \{c_1, \cdots, c_m\}$;
Output: A feasible assignment encoded by $\mathbf{y}$
1  Let $ILP_p$ be the integer linear programming formulation of the problem;
2  Let $LP_p$ be the problem obtained from $ILP_p$ by LP-relaxation;
3  Let $\mathbf{y}^*$ be the optimal solution for $LP_p$;
4  foreach $y_i^*$ in $\mathbf{y}^*$ do
5      if $\forall c \in \mathbf{C}, x_i \notin S_c^-$ then
6          if $y_i^* \geq \frac{1}{3}$ then
7              round $y_i$ to be 1;
8          else
9              round $y_i$ to be 0;
10     if $\forall c \in \mathbf{C}, x_i \notin S_c^+$ then
11         if $y_i^* \leq \frac{2}{3}$ then
12             round $y_i$ to be 0;
13         else
14             round $y_i$ to be 1;
15 return $\mathbf{y}$;
```

Unfortunately this algorithm is incomplete as it does not round such y_i^* where the corresponding x_i presents both nonnegated and negated in different clauses. I have not come up with the plan to deal with this part, as the proof that one rounding strategy of this problem gives feasible solution has some annoying natures.

We hope to prove that after rounding, $\forall c \in \mathbf{C}$, c is still satisfied. To achieve that, nonnegated and negated forms require the rounding direction to be different and their ranges overlap. When x_i presents both forms, it is hard to determine which direction to round y_i into. If we can *transform the oringinal CNF Φ into a new form where no boolean variable x_i presents both negated and nonnegated form in its clauses,* then we can prove that Algorithm 1 provides a feasible solution *with an approximation ratio of 3*, with input being the converted CNF.

Proof.

- **Feasible solution:** For each boolean varibale x_i, it either presents negated form or nonnegated form. Consider any clause c after rounding. Suppose c is not satisfied, then $\forall x_i \in S_c^+$, $y_i^* < 1/3$ and $\forall x_i \in S_c^-$, $y_i^* > 2/3$, then

$$\sum_{x_i \in S_c^+} y_i + \sum_{x_i \in S_c^-} (1 - y_i) < 1,$$

which cannot happen.

- **Approximation ratio:** For y_i^* in $\mathbf{y}^*$, it is increased by a factor of at most 3. Thus,

$$\sum_{i=1}^n y_i \leq 3 \cdot \sum_{i=1}^n y_i^* = 3OPT_{LP} \leq 3OPT_{ILP}$$

Note that this is a minimization problem so the result of LP should be no worse than ILP w.r.t the goal.

□

□

2. (Bonus)Suppose there is a sequence of pearls of different color. Color is denoted as $1 - m$ and the total number of pearls is n. After you read these information and conduct some pre-processing, you need to face lots of of queries.

   A query gives two positions $1 \leq l \leq r \leq n$, and ask whether there exists a color, that at least half of pearls in $[l, r]$ is such color.

   (a) Design a random algorithm to solve this problem. Space complexity of your algorithm should be strictly better than $O(mn)$. Explain your idea briefly, give time complexity for pre-processing and per query, and give space complexity. Your accuray should be better than 99.9%.

   For example, a naive algorithm just read in all pearls as pre-processing. And naively iterate every color and every postion for query. This case, the pre-processing complexity is $O(n)$. For query, it will execute $(r - l) * m$ times, since $r - l$ can achieve $n - 1$, so time complexity per query is $O(mn)$. No extra space needed.

   (Hint: Random choose some color and examine.)

   Solution. I first get a nonrandom algorithm which is simple and should act as a benchmark. The idea is to keep m counters initialized to be 0, one for each color. For pre-processing, fill the sequence into an array, which introduces $\mathrm{O}(n)$ space complexity. When there comes a query, strat traversing from l. For each pearl encountered, add 1 to the counter of its color. When traversing finishes at r, check the counters to see if there is one counter $\geq \frac{r-l+1}{2}$. The total space complexity is $\mathrm{O}(m+n)$, and so is the time complexity per query.

   This simple idea implies that a meaningful random algorithm for this problem should give a time complexity strictly beter than $\mathrm{O}(m+n)$ per query, which dirves us to use some data structure that can provide us with $\mathrm{O}(\log n)$ searching time.

   Inspired by the hint below, we can first establish a Balanced Binary Tree for each color in the preprocessing stage, then for query, randomly pick a color and perform $n_1 =$NumSmallerThan(l) and $n_2 =$NumSmallerThan(r) on its balanced binary tree, then check if $n_2 - n_1 \geq \frac{r-l+1}{2}$. If so, return true; else pick another color randomly from rest of the colors and repeat, until the procedure returns or all color is tried.

```
Algorithm 2: Random Algorithm For Query
   Input: A sequence of pearls $(p_1, \cdots, p_n)$ of different colors
          $\{c_1, \cdots, c_m\}$, two positions $l$, $r$;
1  Initialize balanced binary tree for each color $c$;
2  foreach pearl in the sequence do
3      insert the position of the pearl into the tree of its color;
   // Preprocessing finished
4  foreach color not considered yet do
5      Randomly choose one color(tree) $c$ to check;
6      $n_1 \leftarrow \mathrm{NumSmallerThan}_c(l)$;
7      $n_2 \leftarrow \mathrm{NumSmallerThan}_c(r)$;
8      if $n_2 - n_1 > \frac{r-l+1}{2}$ then
9          return True;
10 return False;
```

   For complexity, this algorithm requires extra $\mathrm{O}(n)$ space to store every pearl as a vertex of the balanced binary tree. The preprocessing part inserts n vertices in total, so it requires $O(n \log n)$ time. For each query, in the worst case every binary tree is visited, each with a scale no larger than $O(n)$, so the worst case time complexity is $O(2m \log n)$.

Note that this is a Las Vegas algorithm, every time we get the correct answer, but the time consumed is a random variable. It is possible that we finish this algorithm in first several visits to the balanced binary trees, which cost $O(c \log n)$ time, where c is a constant. □

(b) **Remark:** This question involves a little bit knowledge about online algorithm. The ddl for this lab is 5/27/2019.

Now there are extra operation besides query.

Append(c): Put a pearl with color c at the end of sequence.

Erase: Take out the last pearl.

Colouration(p,c): Choose pearl of postion p and change its color to c.

Assume that no operation will involve a new color. You may modify your algorithm and show time complexity for each type of operation(include query).

(Hint: Consider Balanced Binary Tree. Given an element e, they can find whether e exists in tree, and how many elements in tree are smaller than e, in $O(logn)$ time.)

Solution. The centural issue for adding these operations might be whether to use another $O(n)$ space complexity to keep a copy of the current sequence in an dynamic array, to speed up the Colouration opeartion. By introducing this array we can quickly locate which tree the pearl to be coloured is oringinally in. Otherwise, we have to randomly search a tree until we find the desired pearl to remove, which may take $O(m \log n)$ in the worst case. We shall compare the two implements in detail.

i. **Not introducing dynamic array**
   - **Append(c):** To make **Erase** efficient, we may want to keep a pointer pointing to the leaf vertices of the last pearl in the current sequence. Actually to append a pearl of some color c is easy, we just need to insert a vertex of value $n + 1$ to the tree of c, which takes $O(\log n)$, and increase n by one. Then we modify the pointer to give it correct value. These other operations take constant time per appending.
   - **Erase:** With help of the pointer, we can easily locate the pearl to erase, decrease n, which takes constant time. But after erasing, we do not know where the current last pearl is. We may have to search m trees to find the last pearl. Adding a pointer to the last but one pearl is the same result, after erasing we lose information on the current last but one pearl and need to search the trees. Therefore **Erase** actually takes $O(m \log n)$ in the worst case as long as we do not intordnce the dynamic array, and on average $O(c \log n)$.
   - **Colouration(p,c):** As has been analyzed, we need to first locate which tree p is in, which may take $O(m \log n)$ in the worst case, delete p in that tree, and insert p in the tree of c. In all on average it will take $O(c \log n)$ per colouration.
   - **Query:** For query(including preprocessing) it is exactly the same as before as no new structure is introduced.

ii. **Introducing dynamic array**
   - **Append(c):** Besides inserting $n + 1$ in the tree of c, which takes $O(\log n)$ time, we also need to insert c into the dynamic to be the new last element. Locating and assigning takes constant time. The problem lies where the dynamic array is full and need to be expanded, this single **Append** takes $O(n)$ time. Luckily

according to amortized analysis, we know that on average each insert into the dynamic array takes constant time. In all each **Append** takes $\mathrm{O}(\log n)$ time.

- **Erase:** With the help of the array this is much more convenient. We get to know which tree the last pearl is in in constant time, and delete the last leaf node of that tree straightly, delete it in the array and decrese n. We may also wish to shrink the size of the array if after some **Erase** the array is 3/4 empty or whatever ratio occupied. By amortized analysis we know that shrinking on average takes constant time per operation. In all it only takes $\mathrm{O}(\log n)$ time for deleting the leaf node, as other operations take constant time.
- **Colouration(p,c):** It shares the same convenience with **Erase** as we can locate which tree p is in in constant time. Delete the oringinal node and insert p into tree of c. Each of the two operation takes $\mathrm{O}(\log n)$ time, so in all $\mathrm{O}(\log n)$ time is used per colouration.
- **Query and Preprocessing:** For preprocessing it introduces extra $\mathrm{O}(n)$ space complexity to keep the array, but in all the total space complexity is still $\mathrm{O}(m+n)$. It also takes $\mathrm{O}(n)$ extra time to fill in n slots of the array, and the total complexity of preprocessing is still $\mathrm{O}(n \log n)$.
  Since query only involves operations with the balanced binary trees and does no modification to the tree, the time complexity remains the same as in (a), that is, $\mathrm{O}(m \log n)$ in the worst case and $\mathrm{O}(c \log n)$ on average.

□

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