Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №8 по курсу «Дискретный анализ»

Студент: И. А. Мариничев
Преподаватель: Н. С. Капралов
Группа: М8О-208Б

Дата: 10.05.21
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №8. Выбор отрезков

Задача: Разработать жадный алгоритм решения задачи, определяемой своим вариантом. Доказать его корректность, оценить скорость и объём затрачиваемой оперативной памяти.

Реализовать программу на языке C или C++, соответсвующую построенному алгоритму. Формат входных и выходных данных описан в варианте задания.

На координатной прямой даны несколько отрезков с координатами $[L_i, R_i]$. Необходимо выбрать минимальное количество отрезков, которые бы полностью покрыли интервал $[0, M]$.

Формат входных данных

На первой строке располагается число **N**, за которым следует **N** строк на каждой из которой находится пара чисел L_i, R_i; последняя строка содержит в себе число **M**.

Формат результата

На первой строке число **K** выбранных отрезков, за которым следует **K** строк, содержащих в себе выбранные отрезки в том же порядке, в котом они встретились во входных данных. Если покрыть интервал невозможно, нужно распечатать число **0**.

1 Описание

Согласно [1], жадные алгоритмы предназначены для решения задач оптимизации. Они обычно представляют собой последовательность шагов, на каждом из которых предоставляется некоторое множестов выборов. В жадном алгоритме всегда делается выбор, который кажется самым лучшим в данный момент, т.е. проводится локально оптимальный выбор в надежде, что он приведет к оптимальному решению глобальой задачи. Важно отметить, что жадные алгоритмы не всегда приводят к оптимальным решениям, но во многих задачах дают нужный результат.

Применим жадный алгортим к задаче выбора отрезков. Сохраним наши пары границ отрезков в массив данных и отсортируем их по правой границе, т.е. при обращении к данному массиву в начале будут лежать отрезки, которые заканчиваются правее всех. Также при сохранении отрезков запомним их изначальный индекс для того чтобы, при выводе восстановить исходный порядок.
Теперь будем считать, что в точке 0, т.е. в начале покрываемого интервала, лежит грань, котрорую мы будем двигать по мере того, как будет происходить покрытие интервала. Эта грань будет означать самую правую точку покрытой части (в начале это точка 0). Пока эта точка не станет $\geq \mathbf{M}$, мы будем считать наш интервал непокрытым.
Затем будем в цикле проходиться по нашему отсортированному масиву и брать первый отрезок, левая граница которого лежит левее или равна текущей грани, а правая граница лежит обязательно правее текущей грани. Это необходимые условия, т.к. мы ищем минимальное число отрезков, иначе есть шанс взять отрезки, котрые не подвинут нашу грань. Если же на каком-то этапе прохода мы не нашли такого отрезка, это говорит о том, что мы не пожем покрыть наш интервал данной выборкой отрезков. Подходящие же отрезки будем сохранять в новый массив, который в случае порытия интервала отсортируем по изначальному индексу и выведем в качестве ответа.
Докажем верность данного алгоритма. Благодаря тому, что на каждом шаге мы берем отрезок с самой правой границей, то это будет гаратировать минимальность количества выбранных отрезков. Здесь важно отметить, что жадные алгоритмы дают нам лишь одно оптимальное решение, которых может быть несколько. Следовательно, допустим, что мы имеем оптимальное решение задачи выбора отрезков и на каком-то этапе мы решаем добавить наш отрезок в это решение. У нас есть два варианта:
1) этот отрезок лежит в этом решении, тогда все ок, просто перейдем к следующй подзадаче;
2) этот отрезок не лежит в этом решении, но т.к. согласно нашему алгоритму на данном этапе мы добавляем отрезок с самой правой границей, то он сдвинет грань еще дальше и обязательно будет иметь пересечние со следующим отрезком в оптимальном решении, значит мы можем просто исключить отрезок лежащий в оптимальном решении на выбранный.

2 Исходный код

Теперь поговорим о реализации данного алгоритма на языке C++. Отрезок представим в виде стркутуры **TClosedInterval**, который будет хранить пару из левой и правой границ и исходный индекс.
Массивом, в котром будут храниться отезки, будет **std::vector**. В прогрмме использованы два вектора: **closedIntervals** и **answer**. В первый складываются отрезки при чтении данных. Во второй будут складываться отрезки, состовляющие оптимальное решение задачи.
Для сортировки веторов воспользуемся **std::sort** и двумя вспомогательными компораторами **decreasingSecondEndpoint** и **compareByOriginalIndex**, которые будут сравнивать отрезки по требуемым параметрам.
Также был введен вспомогательный bool флаг **noAnswer**, который будет сигнализировать об отсутствии решения.

```
1  #include <iostream>
2  #include <utility>
3  #include <vector>
4  #include <algorithm>
5
6  using namespace std;
7
8  struct TClosedInterval {
9      pair<int, int> endpoints;
10     int index;
11 };
12
13 bool decreasingSecondEndpoint(const TClosedInterval &a, const TClosedInterval &b) {
14     return a.endpoints.second > b.endpoints.second;
15 }
16
17 bool compareByOriginalIndex(const TClosedInterval &a, const TClosedInterval &b) {
18     return a.index < b.index;
19 }
20
21 int main() {
22     int n; // number of closed intervals
23     cin >> n;
24
25     TClosedInterval closedInterval;
26     vector<TClosedInterval> closedIntervals; // container for closed intervals
27
28     for (int i = 0; i < n; ++i) { // read N intervals and put them into container
29         std::cin >> closedInterval.endpoints.first >> closedInterval.endpoints.second;
30         closedInterval.index = i; // we are also saving original index of interval
31         closedIntervals.push_back(closedInterval);
32     }
```

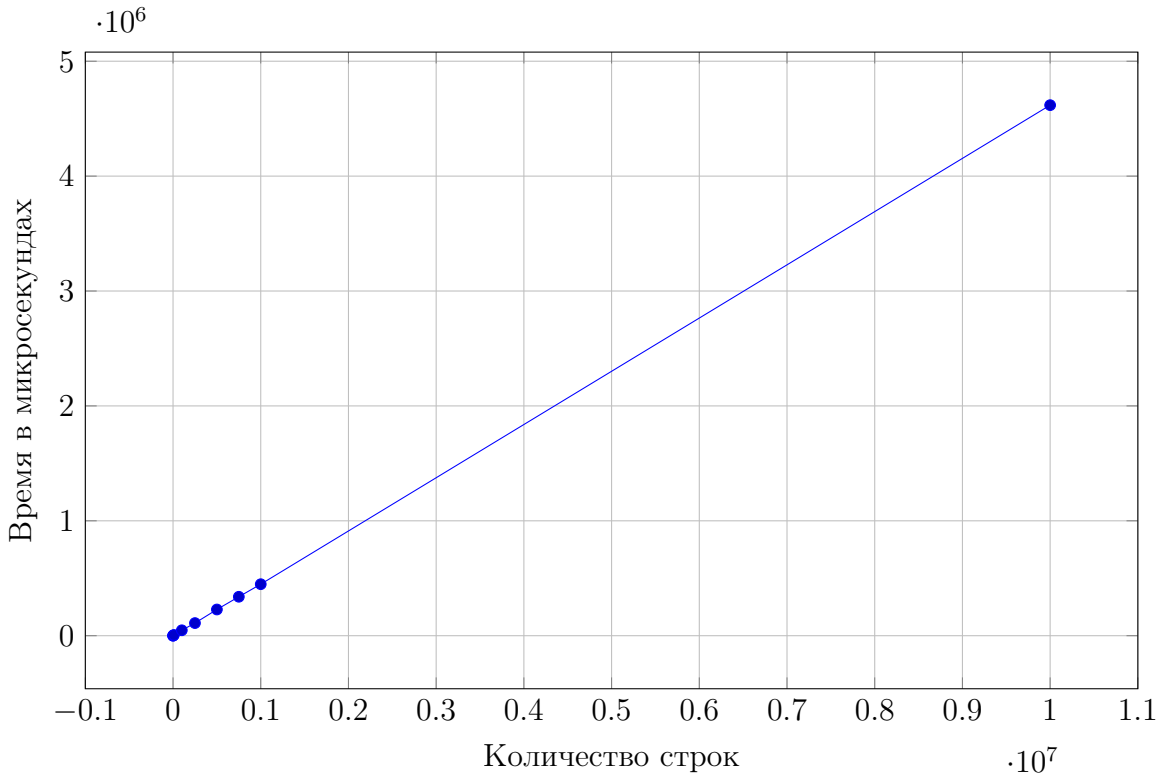
```
33
34      int m; // right endpoint of [0; M] interval
35      cin >> m;
36
37      sort(closedIntervals.begin(), closedIntervals.end(), decreasingSecondEndpoint);
38
39      int k = 0; // amount of taken intervals
40      vector<TClosedInterval> answer; // container for taken intervals
41
42      bool noAnswer = false;
43      int cur = 0; // current rightmost endpoint of covered part of interval
44      while (cur < m) { // while we haven't covered whole interval
45          auto it = closedIntervals.begin();
46          for (; it < closedIntervals.end(); ++it) {
47              // if interval starts at current rightmost endpoint of covered part or
                    earlier
48              // and ends after it
49              if (((*it).endpoints.first <= cur) && ((*it).endpoints.second > cur)) {
50                      answer.push_back(*it); // add interval
51                      k++; // increment counter
52                      cur = (*it).endpoints.second; // update rightmost endpoint of
                            covered part
53                      break; // stop and start looking for the next one
54              }
55          }
56          // iterator will be equal to closedIntervals.end()
57          // if only we iterated through all intrevlas and didn't add anything
58          if (it == closedIntervals.end()) {
59              noAnswer = true; // that means there is no answer
60              break;
61          }
62      }
63
64      if (noAnswer) {
65          cout << 0 << endl;
66          return 0;
67      }
68
69      // print minimum amount of intervals necessary to cover interval [0; M]
70      cout << k << endl;
71
72      sort(answer.begin(), answer.end(), compareByOriginalIndex);
73
74      for (int j = 0; j < k; ++j) { // print taken intervals in their original order
75          cout << answer[j].endpoints.first << " " << answer[j].endpoints.second << endl;
76      }
77
78      return 0;
79  }
```

3 Тест производительности

Теперь прейдем к оценке скорости и объёма затрачиваемой оперативной памяти. Сложность алгоритма O($n * log(n)$), в худшем случае равна O(n^2), если нам нужно будет каждый раз в цикле проходить по всему отсортированному массиву. Объём дополнительной затраченной памяти O(n).
Теперь посмотрим на то, как же ведет себя программа на практке. Проверим время работы программы на тестах с разным количеством строк с отрезками: 100, 1000, 10000, 100000, 250000, 500000, 750000, 1000000, 10000000. Время выводится микросекундах. Для измерения времени использовалась бибилиотека **chrono**.

```
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab8$ make run
./solution <./tests/test_100.t
Time: 186 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab8$ make run
./solution <./tests/test_1000.t
Time: 554 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab8$ make run
./solution <./tests/test_10000.t
Time: 6441 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab8$ make run
./solution <./tests/test_100000.t
Time: 47273 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab8$ make run
./solution <./tests/test_250000.t
Time: 109942 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab8$ make run
./solution <./tests/test_500000.t
Time: 228170 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab8$ make run
./solution <./tests/test_750000.t
Time: 338999 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab8$ make run
./solution <./tests/test_1000000.t
Time: 448275 microsec
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab8$ make run
./solution <./tests/test_10000000.t
Time: 4617539 microsec
```

Представим полученные результаты в более наглядной форме, а именно в виде графика.



С учетом того, что на графике оси имеют разный порядок, можно сделать вывод о том, что время работы программы возрастает логарифмически относительно входных данных, следоваетльно сложность программы действительно равна $O(n*log(n))$.

Тесты создавались с помощью программы на языке Python:

```
from random import choice

NUMBER_OF_LINES = [100, 1000, 10000, 100000, 250000, 500000, 750000, 1000000,
    10000000]

length_of_list = len(NUMBER_OF_LINES)

for j in range(length_of_list):
    with open(f'test_{NUMBER_OF_LINES[j]}.t', 'w') as file:
        file.write(str(NUMBER_OF_LINES[j]) + '\n')
        for i in range(0, NUMBER_OF_LINES[j]):
            l = choice(range(-1000, 1000))
            r = l + (choice(range(1, 10)))
            file.write(str(l) + ' ' + str(r) +'\n')
        file.write(str(choice(range(1000))))
```

4 Выводы

Выполнив восьмую лабораторную работу по курсу «Дискретный анализ», я изучил жадные алгоритмы и применил идею жадного подхода для решения задачи выбора наименьшего числа отрезков для покрытия интевала.

В отличие от динамического программирования жадные алгоритмы предполагают, что задача имеет оптимальное решение, которое строится из оптимальных решений для подзадач с заранее определённым выбором. Такой подход уменьшает временные и пространственные ресурсы, необходимые для решения задачи.

Важно понимать, что жадные алгоритмы применимы не ко всем типам задач, иногда одна из двух, на первый взгляд, похожих задач может решаться при помощи жадного алгоритма, а другая нет. Например, задачи о непрерывном (решается при помощи ЖА) и дискретном рюкзаке (решается при помощи ДП).

Жадные алгоритмы часто используются на практике, т.к. в реальном мире часто приходится работать с данными огромного размера, поэтому вычислительных мощностей для точного алгоритма может не хватать. По этой причине применяются приближенные жадные алгоритмы, которые работают гораздо быстрее и дают лишь одно оптимальное решение.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] *Greedy Algorithms*
URL: `https://www.geeksforgeeks.org/greedy-algorithms/` (дата обращения: 08.05.2021).