ФГБОУ ВО «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

Институт Информационных технологий, математики и механики

Фундаментальная информатика и информационные технологии

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

«Поиск пары пересекающихся отрезков: наивный алгоритм и на основе АВЛ-дерева»

Выполнил:
Кораблев Никита Денисович
Группа: 3821Б1ФИ3

Проверил:

Уткин Герман Владимирович
кафедра: АГДМ

Нижний Новгород
2023

Содержание

1 Введение

В данной лабораторной работе будут расмотрены несколько алгоритмов, решающих задачу поиска пары пересекающихся отрезков. Так же, на основе их реализации и последующего тестирования будет проведена оценка их сложности, а так же сильные и слабые стороны.

2 Постановка задач

Задано множество S, состоящее из n отрезков на плоскости. Каждый отрезок s_i=S[i] (i=1,2,...,n) задан координатами его концевых точек в декартовой системе координат. Требуется определить, есть ли среди заданных отрезков по крайней мере два пересекающихся. Если пересечение существует, то алгоритм должен выдать значение “истина” (“true”) и номера пересекающихся отрезков s_1 и s_2, в противном случае – «ложь» («false»).

2.1 Наивный алгоритм

Наивный алгоритм подразумевает перебор всех отрезков при помощи вложенных циклов до тех пор, пока не будет найденно первое пересечение, или его не будет совсем. То есть, мы смотрим, пересекаются ли отрезки s_i и s_j, где i=(0,1,...,n-1), j=(1,2,...,n), n – кол-во отрезков. И если отрезки пересекаются, то алгоритм завершает свою работу и возвращает найденные пересекающиеся отрезки. И так как мы перебираем отрезки с 0 по n-1 и с 1 по n, то и получаем сложность алгоритма O(n^2).

Пример алгоритма:

```
bool intersectionNaive() {
    int n = S.size();
    bool res = false;
    for (int i = 0; i < n-1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (intersection(S[i], S[j])) {
                std::cout << S[i] << std::endl;
                std::cout << S[j] << std::endl;
                res = true;
                break;
            }
        }
        if (res) break;
    }

    return res;
}
```

2.2 Эффективный алгоритм

В этом алгоритме используется метод вертикальной заметающей прямой, движущейся в сторону возрастания абсциссы. В каждый момент времени заметающая прямая пересекает отрезки, которые образуют динамически меняющееся множество L. Отрезки в множестве L упорядочиваются по неубыванию ординат точек их пересечения с заметающей прямой. Множество L представляется АВЛ-деревом и модифицируется с помощью операций удаления и вставки элементов.

В алгоритме используются две операции getPrev(s) и getNext(s), позволяющие за время O(log(n)) определить непосредственно предшествующий отрезок и отрезок, непосредственно следующий за отрезком s в последовательности L. Если не существует отрезка, предшествующего отрезку s или следующего за ним, то процедуры getPrev(s) и getNext(s) выдают произвольный отрезок, заведомо не пересекающийся с отрезками из множества S.

Операция L.insert(s) осуществляется в момент, когда заметающая прямая достигает левого конца отрезка s, а операция L.remove(s), когда она достигает его правого конца. Обе эти операции осуществляются за время O(log(n)).

Пример алгоритма:

```
bool intersectionEffective() {
    Tree L; bool inter = false;
    segments->sortPoints(0, n-1);
    for (auto p : points) {
        Segment s = segments[p.segmentIndex];
        if (p.isLeft) {
            L.insert(L.getRoot(), s);                          // O(log n)
            Node* addedSegment = L.search(s);                  // O(log n)
            Segment s1 = L.getPrev(addedSegment);              // O(const)
            inter = intersection(s, s1);                       // O(const)
            if (inter) {
                std::cout << s << std::endl;
                std::cout << s1 << std::endl;
                break;
            }
            Segment s2 = L.getNext(addedSegment);              // O(const)
            inter = intersection(s, s2);                       // O(const)
            if (inter) {
                std::cout << s << std::endl;
                std::cout << s2 << std::endl;
                break;
            }
        } else {
            Node* node = L.search(s);                          // O(log n)
            Segment s1 = L.getPrev(node);                      // O(const)
            Segment s2 = L.getNext(node);                      // O(const)
            inter = intersection(s1, s2);                      // O(const)
            if (inter) {
                std::cout << s1 << std::endl;
                std::cout << s2 << std::endl;
                break;
            }
            L.remove(s);                                       // O(log n)
        }
    }
    return inter;
}
```

Так как множество L это AVL-дерево, то операции вставки и удаления работают за O(log(n)), где n – кол-во элементов в дереве. Поиск в АВЛ-дереве – O(log(n)). Операции getPrev(s) и getNext(s), как было описано ранее, должны работать за время O(log(n)), так как внутри самих операций сначала происходит поиск отрезка s в дереве за O(log(n)) и потом получение предыдущего или следующего за O(const). Но в целях оптимизации, операция поиска была вынесена за действие обоих методов. Следовательно последовательный вызов операций search(), getPrev() и getNext() в сумме будут иметь сложность O(log(n)).

Таким образом асимптотическая сложность данного алдгоритма O($n * log(n)$).

3 Руководство пользователя

3.1 Использование готовых тестов

Для запуска готовых тестов, необходимо в main.cpp файле импортировать библиотеку "Tests.h"и в теле main функции вызвать тесты: test1(fileDir), test2(fileDir), test3(fileDir), test4(fileDir), где fileDir - это полгый путь до .txt файла, куда будет сохраняться время работы алгоритмов.

Пример записи данных в файл для тестов 1, 2 и 3:
1 23 5120 14
101 28 5039 7
201 36 4627 6
301 33 4471 3
401 42 4447 3
...

1ый столбец - количество отрезков в множестве, 2ой - время работы наивного алгоритма, 3ий - время сортировки массива точек, и 4ый - время работы эффективного алгоритма. Четвертый тест, в 1ый столбец запишет длину отрезков всего множества.

Также, для получения "настоящих"рандомных значений, в начале файла следует написать: srand(time(NULL)).

3.2 Вызов алгоритмов вручную

Для использования алгоритмов без тестов, достаточно вызвать библиотеку "Segments.h". После чего, создайте объект для хранения отрезков - Segs s;, и заполните его отрезками. Тепрь можно вызвать либо наивный алгоритм:

<multiple segments>.intersectionNaive();
и эффективный:

<multiple segments>.sortPoints(0, <multiple segments>.getPointLen()-1);
<multiple segments>.intersectionEffective();

Для создания точек, используйте:

Point <point name>(x, y);
Для создания резков, используйте:

Segment <segment name>(<point 1>, <point 2>);
Чтобы добавить отрезок в множество отрезков, используйте ТОЛЬКО:

<multiple segments>.pushBack(<segment name>);

4 Руководство программиста

4.1 Структура проекта

PSeg.h - прототипы точки вектора и сегмента, PSeg.cpp - их реализация.
Tree.h - прототип АВЛ-дерева на основе отрезков, Tree.cpp - его реализация.
Segments.h - прототип множества сегментов, Segments.cpp - его реализация.
Tests.h - прототип тестов для замера времени работы наивного и эффективного алгоритмов, Tests.cpp - их реализация.
main.cpp - запускаемый проект для экспериментов.

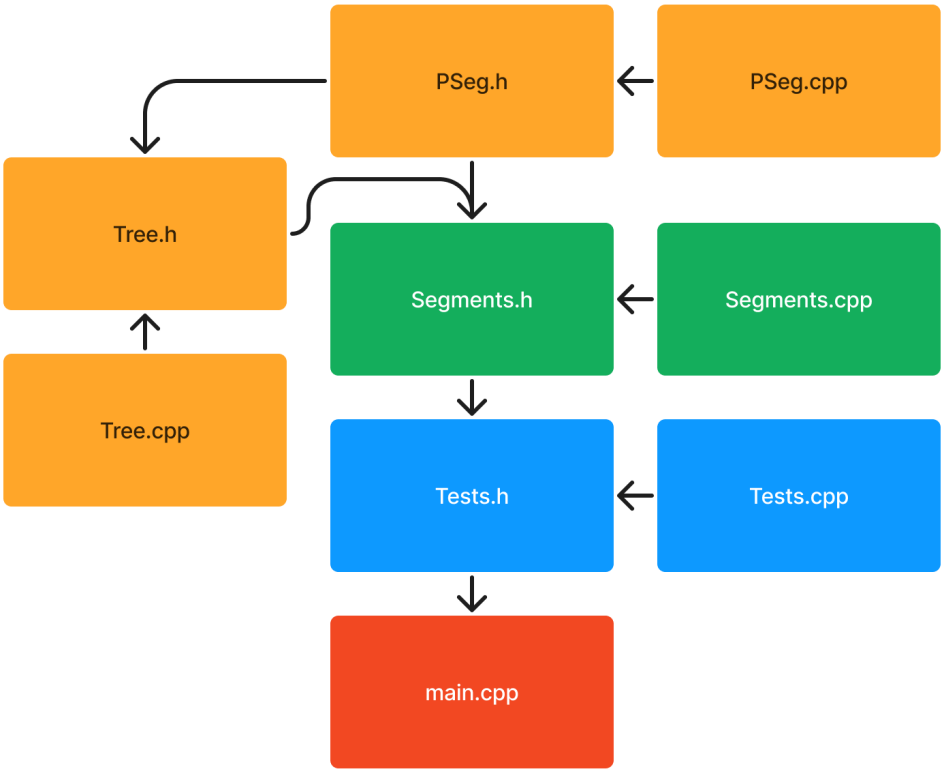


Рис. 1: Схема проекта

5 Тестирование

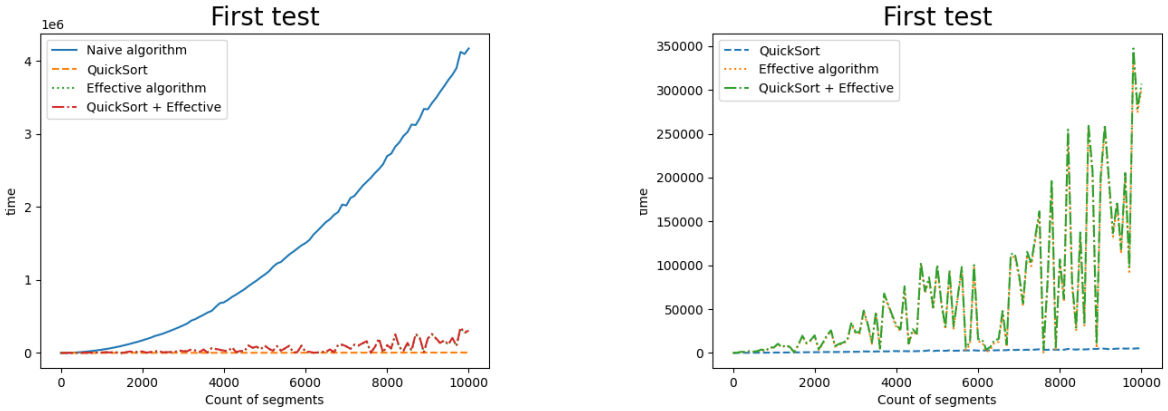
5.1 Аппратные характеристики

- Ноутбук: Lenovo Legion 7 15IMH05
- Процессор: Intel® Core™ i7-10750H CPU @ 2.60GHz × 12 (разогнан до 4GHz)
- RAM: DDR4 - 2x 8ГБ
- OS: Ubuntu 22.04.3LTS
- gcc: (Ubuntu 11.4.0-1ubuntu1-22.04) 11.4.0

5.2 Результаты

Тест 1

Первый способ задания отрезков: n = 1, ..., 10^4, с шагом 100.



(a) With naive algorithm

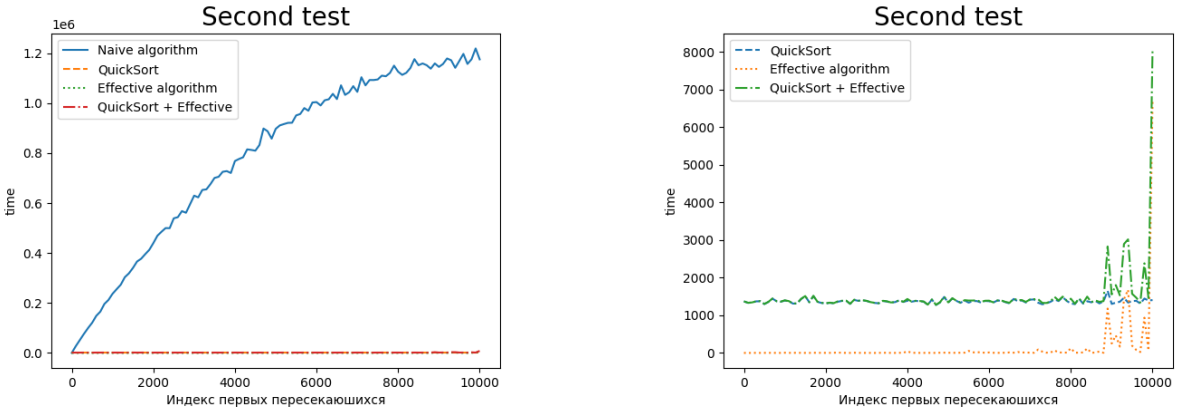
(b) Without naive algorithm

Рис. 2: Зависимость времени работы алгоритмов от количества не пересекающихся отрезков

По сути, данный пример показывает худший вариант, когда во всем множестве отрезков нет ни одного пересечения, и сложность работы наивного алгоритма (Рис. 2) равна $O(n^2)$. Тем временем, график работы еффективного алгоритма показывает время работы $O(n * log(n))$.

Тест 2

Второй способ задания отрезков: n = $10^4 + 3$, k = 1,. . . ,$10^4 + 1$, с шагом 100.



(a) With naive algorithm

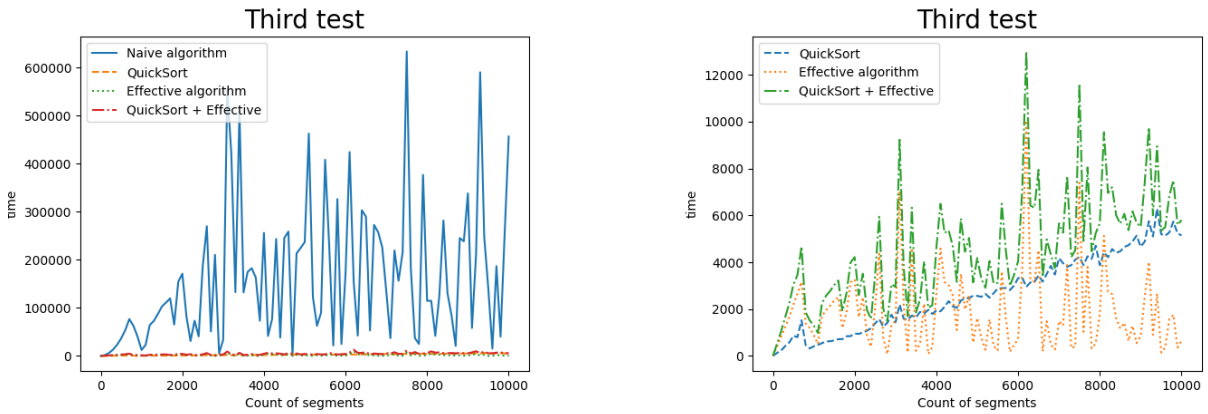
(b) Without naive algorithm

Рис. 3: Зависимость времени работы алгоритмов от места расположения 2ух пересекающихся отрезков.

По скольку наивный алгоритм сравнивает попарно все отрезки в множестве, то время его работы напрямую зависит от расположения этих пересекающихся отрезков с индексами k и k+1. Но на время работы эффективного алгоритма, это ограничение не влияет.

Тест 3

Третий способ задания отрезков: r = 0.001, n = 1, ..., $10^4 + 1$, с шагом 100.



(a) With naive algorithm

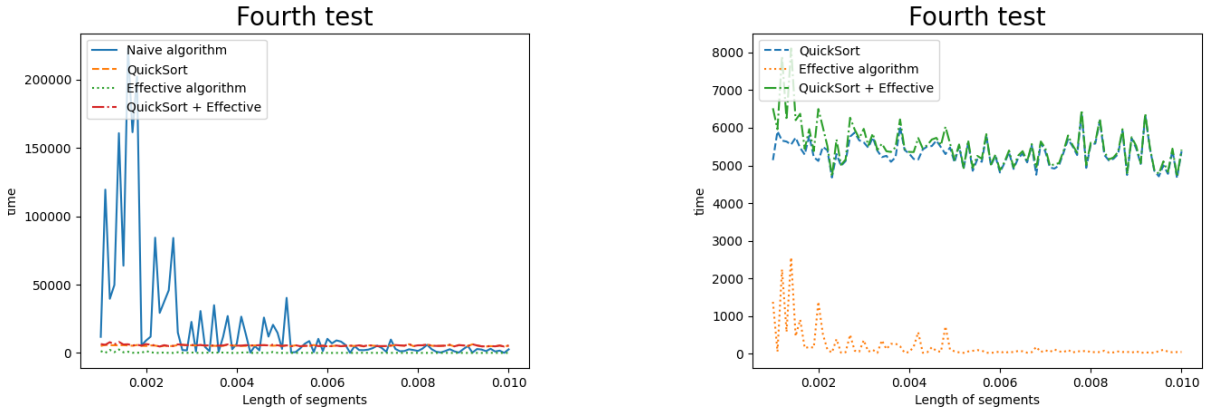
(b) Without naive algorithm

Рис. 4: Зависимость времени работы алгоритмов числа отрезков размера r.

В отличие от первых двух тестов, где пересекающихся отрезков либл не было, либо их было 2, и они постепенно перемещались от начала множества к его концу, количество и расположение в множестве пересекающихся отрезков полностью случайно. Следовательно, мы получаем такой разброс времени работы наивного алгоритма.

Тест 4

Четвертый способ задания отрезков: r $= 1 * 10^{-4}$, ..., 0.01 с шагом $1 * 10^{-4}$, n=10^4.



(a) With naive algorithm

(b) Without naive algorithm

Рис. 5: Зависимость времени работы алгоритмов от размера отрезков.

Из графиков видно, что после достижения отрезками некоторой длинны, время работы наивного алгоритма начинает уменьшаться. А эффективный алгоритм почти всегда работает за константное время. Так же можно заметить, что всплески на графике эффективного алгоритма совпадают с всплесками наивного.

6 Заключение

Можно сделать вывод, что стоит тратить усилия на разработку алгоритма эффективного поиска пересечений, только если мы имеем относительно большое множество отрезком малой длинны (первые три теста). В остальных случаях, за счет простоты реализации, наивный алгоритм будет более эффетивным. Так же, слабым местом эффективного алгоритма, является сортировка.

7 Список Литературы

[1] Семинар 8. Метод заметающей прямой (Алгоритмы и структуры данных, часть 1). URL: https://www.youtube.com/watch?v=sINIi2mwYls&t=481s&ab_channel=ComputerScienceCenter/

[2] GeeksforFeeks. URL: https://www.geeksforgeeks.org/introduction-to-avl-tree/?ref=lbp

[3] ХАБР. URL: https://habr.com/ru/articles/267037/

[4] Моя программа. URL: https://github.com/NikitaKorablev/Intersections_of_segments