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:
Студент
Баранов Алексей Вадимович Группа: 3822Б1ФИ1

Проверил:
Ассистент каф. АГДМ
Уткин Герман Владимирович

Нижний Новгород
2024

Содержание

Введение

В данной работе будут исследованы 2 способа решения задачи поиска пересекающихся отрезков. Так же, на основе их реализации и тестирования на различных данных будет проведена оценка их сложности, а также сильные и слабые стороны.

Постановка задачи

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 s1 и s2, в противном случае “ложь” (“false”).

Наивный алгоритм поиска пересечения

Наивный алгоритм подразумевает перебор всех отрезков при помощи вложенных циклов до тех пор, пока не будет найдено 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). Функция возвращает ответ на поставленный вопрос, и, если ответ положительный, в двух входных параметрах возвращаются уникальные номера отрезков, по которым можно однозначно определить пересекающиеся отрезки.

Пример алгоритма:

```
bool find_intersect(size_t& f, size_t& s) {
  auto size = segs.size();

  for (int i = 0; i < size - 1; ++i) {
    for (int j = i + 1; j < size; ++j) {
      if (intersect(segs[i], segs[j])) {
        f = segs[i].loc_id;
        s = segs[j].loc_id;
        return true;
      }
    }
  }
  return false;
}
```

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 используется метод find_segs_for_checking(), который в определенной точке x перестраивает дерево, в std::vector ctd записывает указатели на вставленный отрезок, его верхнего соседа и нижнего(если они есть). Методы поиска соседей в AVL дереве работают за O(log(n)). Операция tree.insert() осуществляется в момент, когда заметающая прямая достигает левого конца отрезка s, а операция tree.remove(s), когда она достигает его правого конца.

Пример алгоритма:

Основная функция:

```
bool find_intersect(size_t& f, size_t& s) {
  sort points();//std::sort()
  ctd.resize(3);

  for (auto&& point_curr : points)  // 2n instances
  {
    if (!point_curr.ref_to_line->is_in_tree) {
      point_curr.ref_to_line->is_in_tree = true;
      tree.insert(point_curr.y, *point_curr.ref_to_line);//O(log(n))

      tree.find_segs_for_checking(point_curr.x, point_curr.y, ctd); //O(log(n))

      if (ctd[1] != nullptr) {
        if (intersect(*ctd[0], *ctd[1])) { //O(const)
          f = ctd[0]->loc_id;
          s = ctd[1]->loc_id;
          return true;
        }
      }
      if (ctd[2] != nullptr) {
        if (intersect(*ctd[0], *ctd[2])) { //O(const)
          f = ctd[0]->loc_id;
          s = ctd[2]->loc_id;
          return true;
        }
      }
    } else {
      tree.find_segs_for_checking(point_curr.x, point_curr.y, ctd);
      if (ctd[1] != nullptr && ctd[2] != nullptr) {
        if (intersect(*ctd[1], *ctd[2])) { //O(const)
          f = ctd[0]->loc_id;
          s = ctd[1]->loc_id;
          return true;
        }
      }
      tree.remove(point_curr.y); // O(log(n))
    }
  }
  return false;
};
```

Дополнительная функция:

```
void find_segs_for_checking(double x, double key, std::vector<Data*>& vec) {
  re_calculate_tree_nodes_at_point(x);
  using it = TreeNode*;
  it look_at_loc = search(key);//O(log(n))
  if (look_at_loc != nullptr) {
    vec[0] = &look_at_loc->value;

    it upper = find_t(look_at_loc);//O(log(n))
    it lower = find_d(look_at_loc);//O(log(n))
    vec[1] = (upper) ? &upper->value : nullptr;
    vec[2] = (lower) ? &lower->value : nullptr;
  }
}
```

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 `find_t` (s) и `find_d` (s), должны работать за время O(log(n)), внутри них происходит поиск соседей указанного отрезка и потом получение предыдущего или следующего за O(const). В целях оптимизации, операция поиска была вынесена за действие обоих методов. Следовательно последовательный вызов операций search(), find_t() и find_d() в сумме будут иметь сложность O(log(n)).

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

Руководство пользователя

Использование готовых тестов

Для запуска готовых тестов, необходимо в аргументах командной строки передать одну из следующих строк:

// Каждая из них запускает выполнение тестов от 1 до 4 по порядку.

autogen_version

autogen_version k_varies

autogen_version length_test

autogen_version length_varies

Пример записи данных в файл для тестов:

```
 1 -> 0 0
101 -> 0 0
        …
```

2ой столбец - время работы эффективного алгоритма,
3ий - время работы наивного алгоритма.
1 столбец показывает различные вещи в зависимости от “режима” тестирования алгоритма: 1 и 3 тесты – количество отрезков, 2 – число k первых непересекающихся отрезков, 4 – длины отрезков, которые проверяются на пересечение. Каждому тесту соответствует свой выходной файл с именем outstream<test_number>.txt в директории source_folder.

Пользовательское тестирование

Для написания собственных тестов пользователю достаточно собрать проект с дополнительным define:

-D visual

С использованием этого флага на консоль будет печататься вся необходимая пользователю информация.

И запустить с любым из аргументов:

sample.txt

sample2.txt

В таком режиме работы программа будет читать вышеперечисленные файлы и интерпретировать их содержимое как исходные данные, суть отрезки.

Пример задания отрезков в файле:

(1,1) (2,2)

(3,3) (4,4)

Руководство программиста

Структура проекта

Tree.hpp – реализация AVL-дерева.

Header.hpp – файл заголовков, где хранятся структуры данных и методы решения основной задачи.

Tests_header.hpp – реализации функций генерирования случайных отрезков

Source_func_for_main.hpp – реализации основных функций, вызываемых в main.

source.cpp - запускаемый проект для экспериментов.

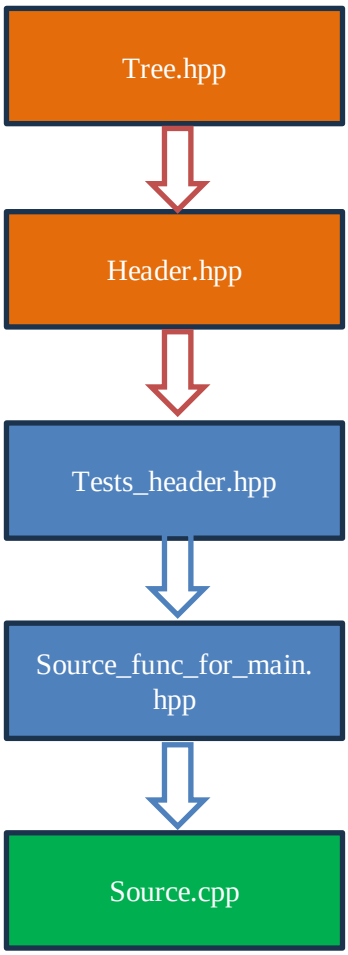

Рис. 1: Схема проекта

Тестирование

Аппаратные характеристики

- Ноутбук: HP Laptop 15s-fq2xxx
- Процессор: 11th Gen Intel(R) Core(TM) i3-1125G4 @ 2.00GHz
- RAM: DDR4 8ГБ
- OS: Windows 11
- msvc: 19.3x

Результаты

Тест 1

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

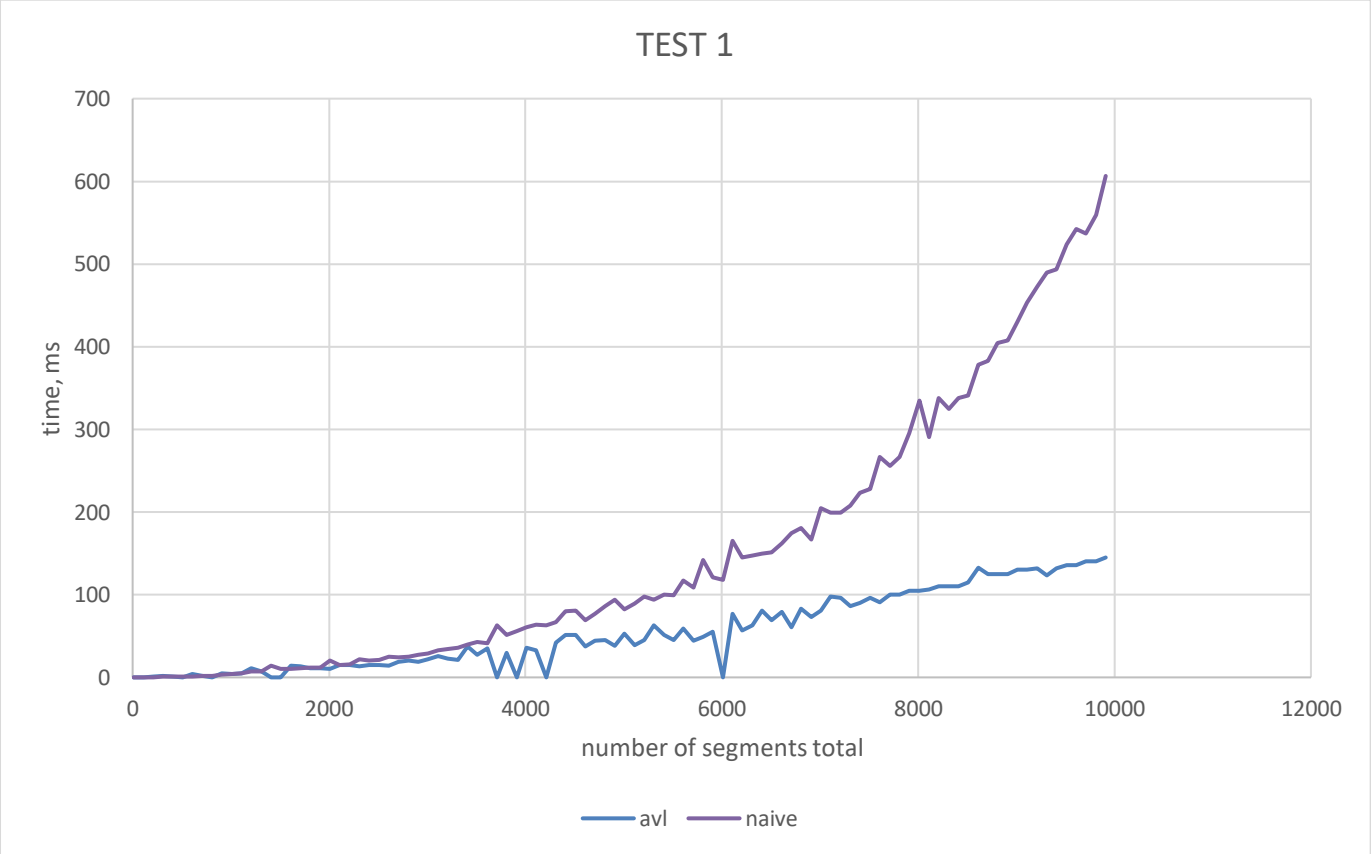

Рис. 2: Зависимость времени работы алгоритмов от количества не пересекающихся отрезков

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 эффективного алгоритма показывает время работы $O(n * log(n))$.

Тест 2

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

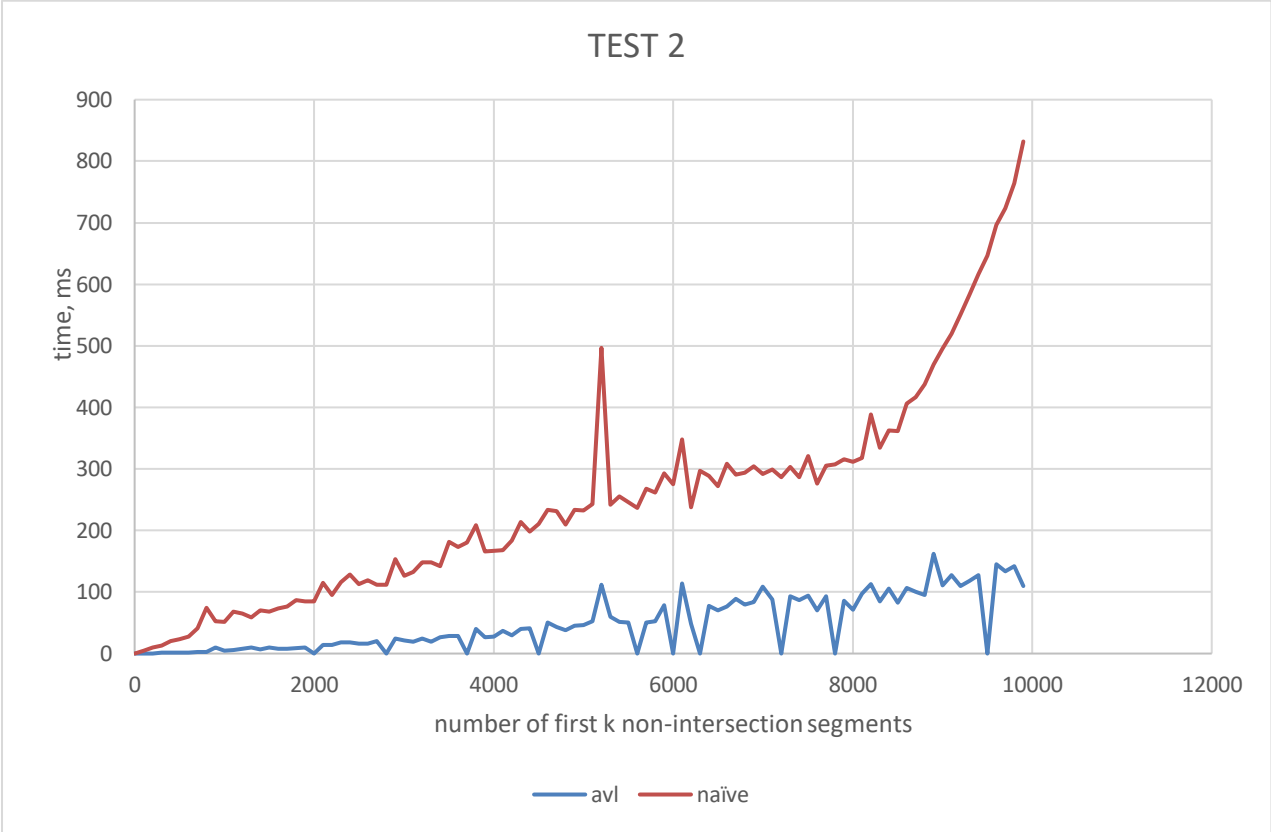

Рис. 3: Зависимость времени работы алгоритмов от места расположения 2ух пересекающихся отрезков.

Поскольку 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.

Тест 3

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

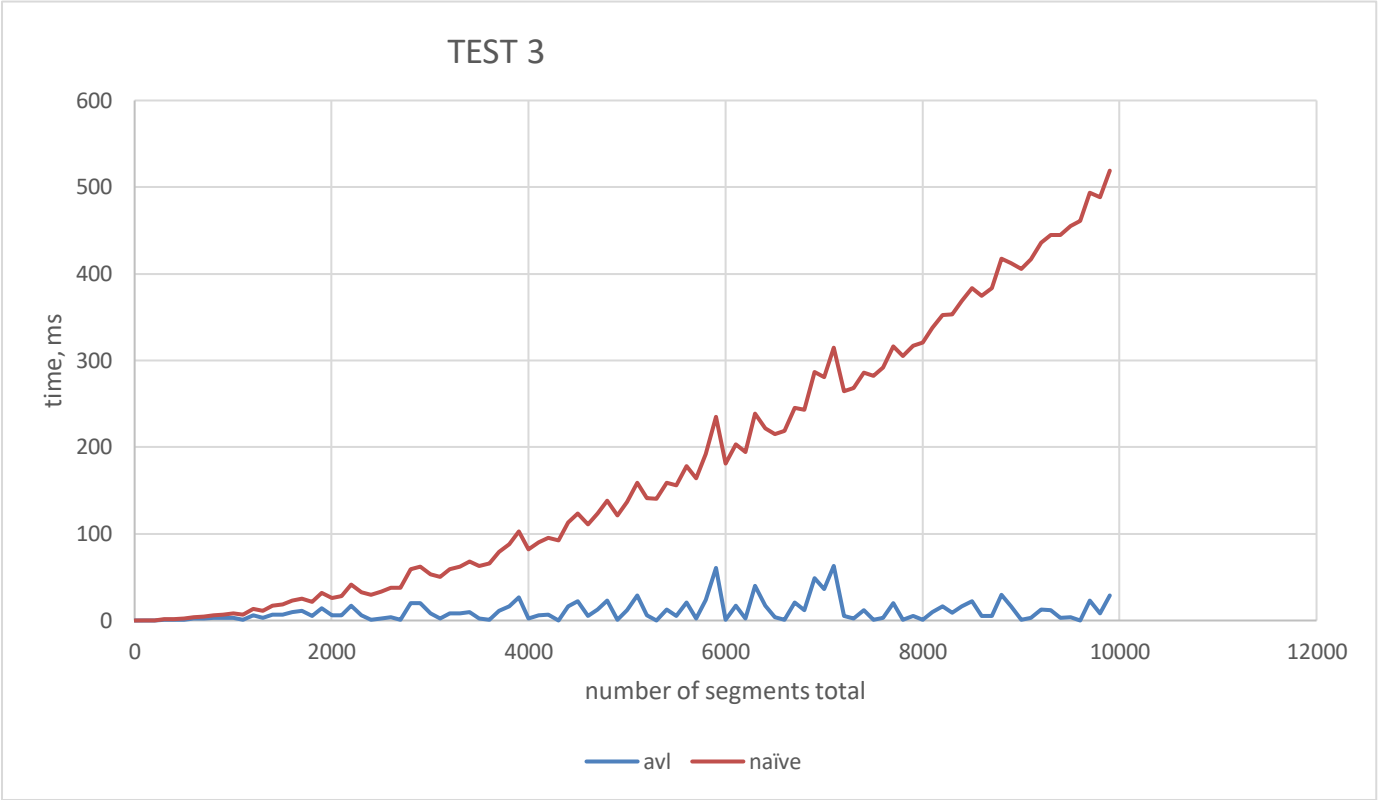

Рис. 4: Зависимость времени работы алгоритмов числа отрезков размера r.

В отличие от первых двух тестов, где пересекающихся отрезков либо 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.

Тест 4

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

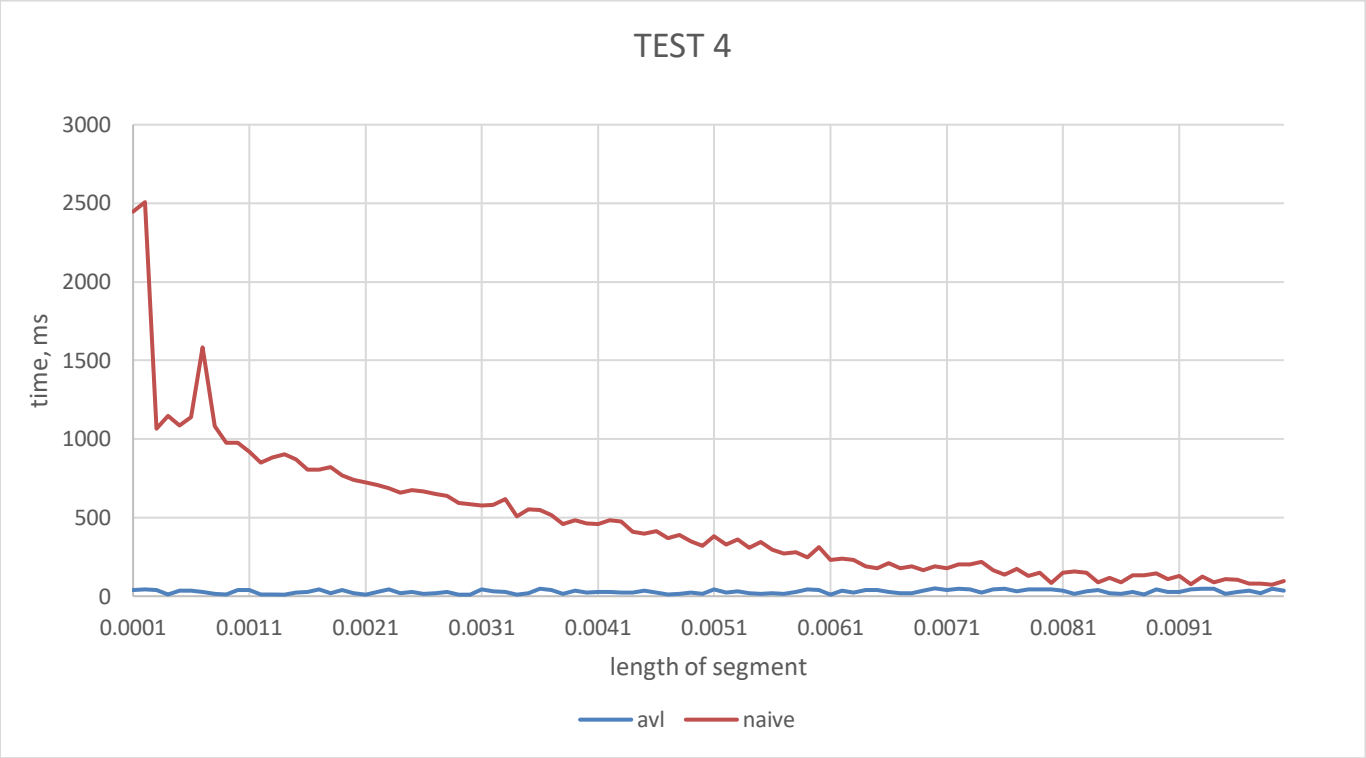

Рис. 5: Зависимость времени работы алгоритмов от размера отрезков.

Из графиков видно, что после достижения отрезками некоторой длинны, время работы наивного алгоритма начинает уменьшаться. А эффективный алгоритм почти всегда работает за константное время.

Заключение

Подводя итоги проведенной работы, можно утверждать, что разработка эффективного алгоритма имеет смысл только в случаях, когда необходимо проверять довольно большое число отрезков, потому что наивный алгоритм показывает себя вполне конкурентноспособным на небольшом числе отрезков, тем более , что разработка простого алгоритма на порядок проще, по сравнению с алгоритмом на основе AVL дерева

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] Метод заметающей прямой: https://neerc.ifmo.ru/wiki/index.php

[4] https://www.graphicon.ru/html/2010/conference/RU/Se4/021.pdf

[5] Алгоритмы вычислительной геометрии. Пересечение отрезков: метод заметания плоскости. Ивановский Сергей Алексеевич, Симончик Сергей Константинович 2007.