МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский Нижегородский государственный университет

им. Н.И. Лобачевского»

Институт информационных технологий математики и механики

Отчёт по лабораторной работе на тему:

Поиск пары пересекающихся отрезков: на основе АВЛ-дерева и (2, 3)-дерева

Выполнила: студент группы 3822Б1ФИ1
Направление 02.03.02 «Фундаментальная информатика и информационные технологии»
Бесхмельнова Ксения

Преподаватель: Ассистент каф. АГДМ Уткин Герман Владимирович

Нижний Новгород

2024

Содержание

Введение

Поиск пары пересекающихся отрезков является одной из ключевых задач вычислительной геометрии, с широким применением в компьютерной графике, геоинформационных системах и моделировании физических процессов. Эффективное решение этой задачи требует использования подходящих структур данных, которые обеспечивают оптимальное время выполнения. В данной лабораторной работе исследуется применение АВЛ-дерева и (2, 3)-дерева для решения задачи поиска пары пересекающихся отрезков. Приведены теоретические основы использования этих структур данных, а также проведён сравнительный анализ их производительности на различных наборах данных.

1 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 s_1 и s_2, в противном случае – «ложь» («false»).

1.1 Наивный алгоритм

При использовании наивного алгоритма перебираются пары отрезков до тех пор, пока не будет обнаружено пересечение, либо же не будут исчерпаны все пары.

Псевдокод:

```
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 ;
}
```

Внешний цикл выполняется n−1 раз.
Внутренний цикл на каждой итерации выполняется (n−i−1) раз, где i — индекс текущей итерации внешнего цикла.
Например:

- На первой итерации внешнего цикла (i = 0) внутренний цикл выполнится n – 1 раз.
- На второй итерации внешнего цикла (i = 1) внутренний цикл выполнится n – 2 раз.
- И так далее.

Общее количество сравнений — сумма чисел от 1 до (n−1):

$$(n-1)+(n+2)+\cdots+2+1=\frac{n(n-1)}{2},$$

что асимптотически эквивалентно $O(n^2)$.

1.2 Эффективный алгоритм

В данном алгоритме используется подход с заметающей прямой, которая движется слева направо, увеличивая значение координаты x. В каждый момент времени прямая пересекает множество L, состоящее из отрезков, расположенных по возрастанию их пересечения с заметающей прямой. Множество L организуется в виде структуры данных, такой как АВЛ-дерево или (2,3)-дерево, что позволяет эффективно обновлять его при добавлении или удалении отрезков.

Для работы алгоритма применяются две ключевые операции: getPrev(s) и getNext(s). Эти операции позволяют за O(log(n)) находить отрезок, непосредственно предшествующий s, или отрезок, следующий за s в множестве L. Если таких отрезков нет, методы возвращают отрезок, который гарантированно не пересекается с другими отрезками из L.

Добавление отрезка s в L через L.insert(s) происходит, когда заметающая прямая достигает левого конца отрезка. Аналогично, удаление отрезка s через L.remove(s) осуществляется в момент достижения правого конца отрезка. Оба действия выполняются за O(log(n)), благодаря использованию эффективной структуры данных.

Таким образом, организация множества L с использованием деревьев позволяет существенно повысить производительность алгоритма по сравнению с менее оптимальными подходами.

Псевдокод:

```
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);
          Node∗ addedSegment = L. search ( s );
          Segment s1 = L.getPrev (addedSegment);
         inter = intersection (s , s1 );
         if ( inter ) {
             std : : cout << s << std : : endl ;
             std : : cout << s1 << std : : endl ;
               break;
          }
          Segment s2 = L. getNext (addedSegment);
         Node∗ node = L.search (s);
         Segment s1 = L.getPrev (node);
         Segment s2 = L.getNext (node);
```

```
            inter = intersection (s1 , s2 );
            if ( inter ) {
                std : : cout << s1 << std : : endl ;
                std : : cout << s2 << std : : endl ;
                break;
            }
            L. remove(s);
        }
    }
    return inter;
}
```

Так как множество L представлено в виде АВЛ-дерева или (2, 3)-дерева, основные операции, такие как вставка и удаление элементов, выполняются за O(log(n)), где n — текущее количество элементов в дереве. Аналогично, поиск элемента в дереве также имеет сложность O(log(n)).

Операции getPrev(s) и getNext(s) позволяют находить отрезки, непосредственно предшествующие или следующие за s, за O(log(n)). Эти операции включают поиск отрезка s в дереве (выполняемый за O(log(n))), после чего нахождение предыдущего или следующего отрезка выполняется за O(1).

Для повышения эффективности алгоритма поиск отрезка s был вынесен за рамки методов getPrev и getNext. Это позволяет последовательному вызову операций search(), getPrev() и getNext() сохранять общую сложность O(log(n)).

Таким образом, асимптотическая сложность алгоритма в целом составляет O(n·log(n)), где n — общее количество отрезков.

2 Руководство пользователя

2.1 Запуск программы

Исполняемый файл программы принимает 2 аргумента командной строки:

1. Номер теста: от 1 до 4.
2. Тип алгоритма: naive, effective_AVLTree или effective_Tree23.

Пример запуска программы:

./intersect_segments 1 effective_AVLTree

Этот вызов выполнит тест 1 с использованием алгоритма на основе АВЛ-дерева, а результаты будут записаны в файл results/test1_effective_AVLTree.csv.

2.2 Описание тестов

Тест 1:

- Генерируются случайные отрезки с произвольными координатами.
- Измеряется время выполнения алгоритма для количества отрезков от 100 до 10,000 с шагом 100.

Тест 2:

- Первая часть отрезков генерируется так, чтобы не пересекаться между собой. Два дополнительных отрезка гарантированно пересекаются.
- Измеряется время выполнения алгоритма при увеличении k.

Тест 3:

- Генерируются отрезки фиксированной длины r=0.001 с центрами в случайных точках.

- Измеряется время выполнения алгоритма для увеличивающегося количества отрезков от 100 до 10,000 с шагом 100.

Тест 4:

- Количество отрезков фиксировано (n=10,000), их длина варьируется от r=0.0001 до r=0.01 с шагом 0.0001.
- Измеряется время выполнения алгоритма для каждой длины отрезков.

2.3 Результаты тестов

Каждый тест создает файл формата CSV с результатами. Имя файла формируется как <test><номер теста>_<тип алгоритма>.csv.
Например, results/test1_effective_AVLTree.csv
Структура файла CSV:

- Первый столбец: Значение параметра (например, количество отрезков n или значение r).
- Второй столбец: Время выполнения алгоритма в секундах.

3 Руководство программиста

3.1 Структура проекта

Основные файлы разделены на три категории: исходные файлы, файлы заголовков, и файлы ресурсов. Структура проекта представлена следующим образом:

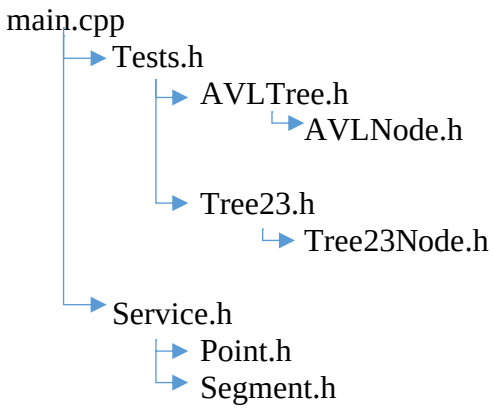
```
├── Исходные файлы
│   ├── main.cpp
├── Файлы заголовков
│   ├── AVLNode.h        // Узел АВЛ-дерева
│   ├── AVLTree.h        // Реализация АВЛ-дерева
│   ├── Point.h          // Структура точек
│   ├── Segment.h        // Структура отрезков
│   ├── Service.h        // Вспомогательные функции
│   ├── Tests.h          // Интерфейс для тестирования алгоритмов
│   ├── Tree23.h         // Реализация (2,3)-дерева
│   ├── Tree23Node.h     // Узел (2,3)-дерева
├── Файлы ресурсов
│   ├── Point.cpp        // Реализация функций работы с точками
│   ├── Segment.cpp      // Реализация функций работы с отрезками
│   ├── Service.cpp      // Реализация вспомогательных функций
```

Описание файлов

1. **Исходные файлы**
   - main.cpp: содержит точку входа программы. В этом файле вызываются функции тестирования алгоритмов для работы с АВЛ-деревом и (2,3)-деревом. Также здесь обрабатываются параметры командной строки.

2. **Файлы заголовков**
    - AVLNode.h и AVLTree.h: реализуют узлы и операции над АВЛ-деревом, включая вставку, удаление, поиск, а также методы getPrev() и getNext() для определения соседних отрезков.
    - Tree23Node.h и Tree23.h: описывают структуру узлов и операций над (2,3)-деревом, включая балансировку, вставку, удаление и поиск.
    - Point.h и Segment.h: определяют базовые структуры данных, такие как точки (абсцисса и ордината) и отрезки.
    - Service.h: содержит вспомогательные функции (например, сортировка точек, проверка пересечения отрезков).
    - Tests.h: интерфейс для запуска тестов. Реализует функции генерации тестовых данных и вызова алгоритмов.
3. **Файлы ресурсов**
    - Point.cpp: реализация методов для работы с точками, включая сортировку и сравнение.
    - Segment.cpp: реализация методов для работы с отрезками, включая определение пересечений.
    - Service.cpp: реализация утилитарных функций, используемых в процессе выполнения алгоритмов и тестов.

3.2 Схема проекта



— **main.cpp** вызывает функции из Tests.h для тестирования алгоритмов.
— **Tests.h** использует структуры данных, определенные в AVLTree.h и Tree23.h.
— **Service.h** обеспечивает вспомогательные функции для обработки данных (например, сортировку точек и проверку пересечений).

- **Файлы ресурсов (*.cpp)** содержат реализацию соответствующих заголовков (*.h).

4 Тестирование

4.1 Аппратные характеристики

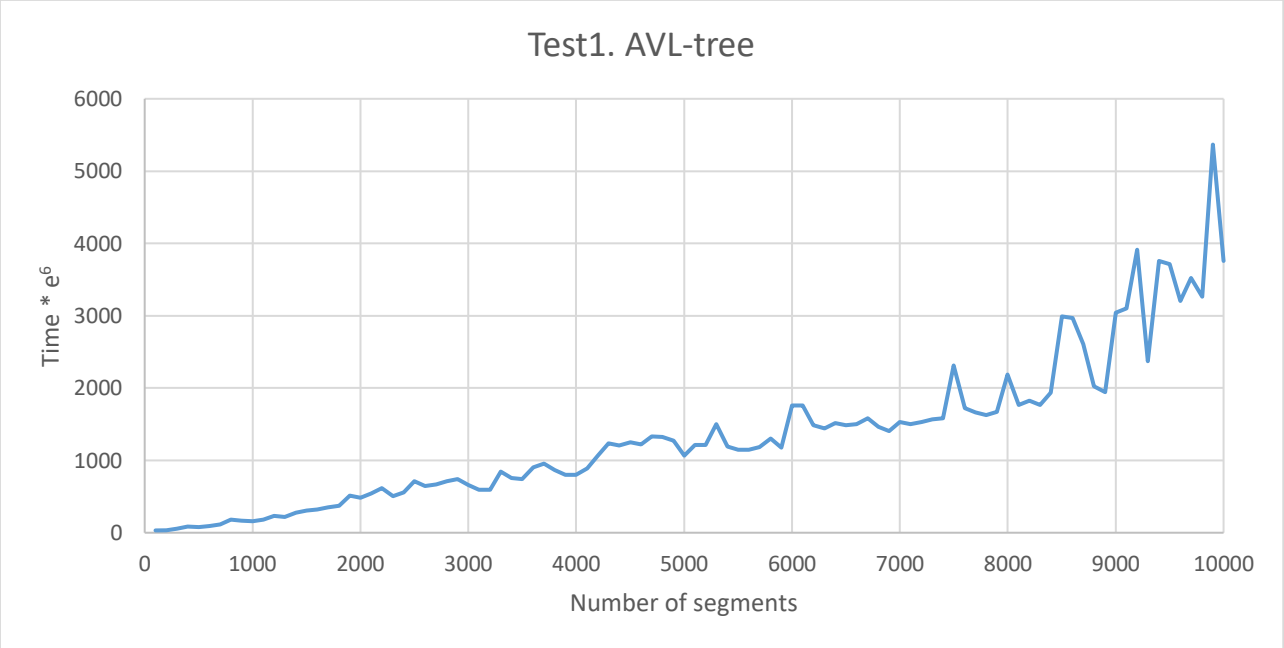
- Ноутбук: HUAWEY Model NBD-WXX9-PCB-B4
- Процессор: Intel Core i5-1135G7 CPU @ 2.40GHz

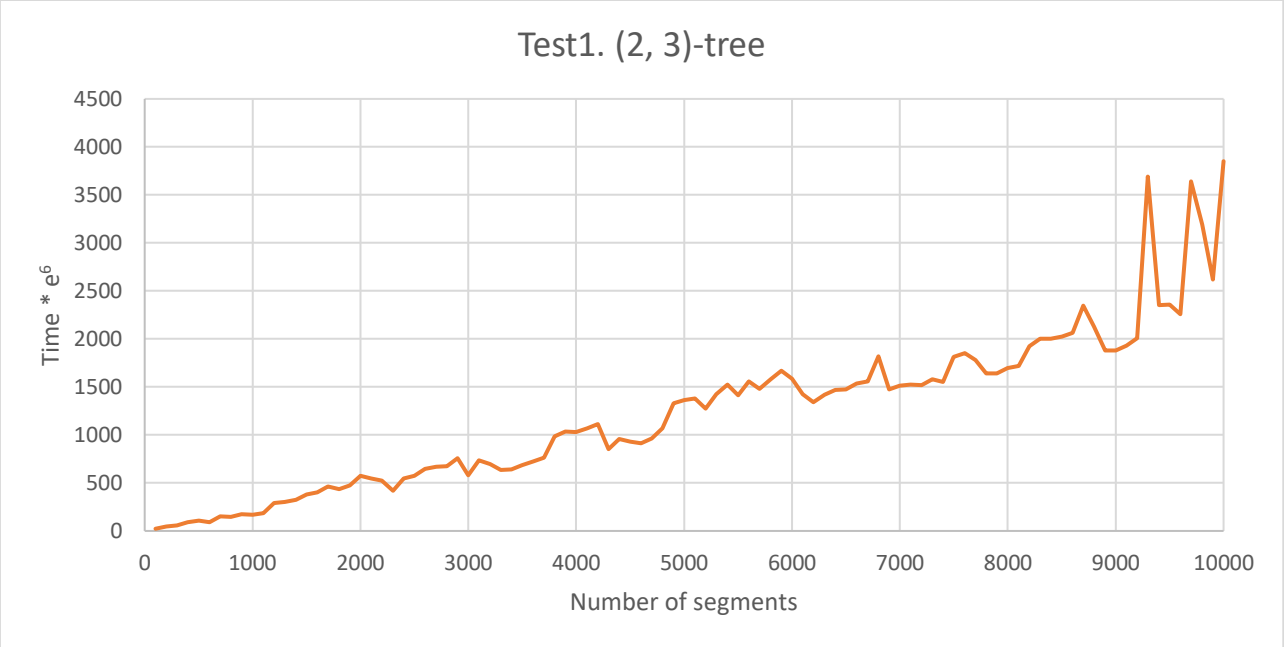
- RAM: DDR4 - 8ГБ
- OS: Windows 11 Version 23H2

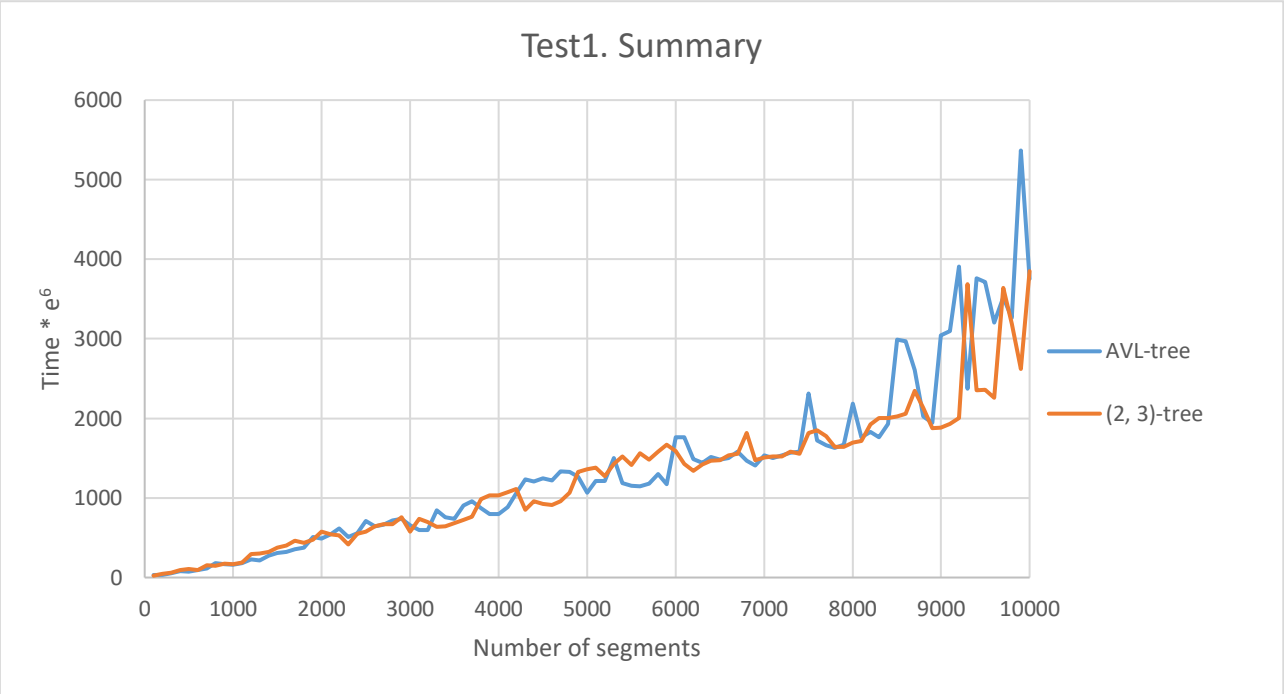
4.2 Результаты экспериментов

Тест 1

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



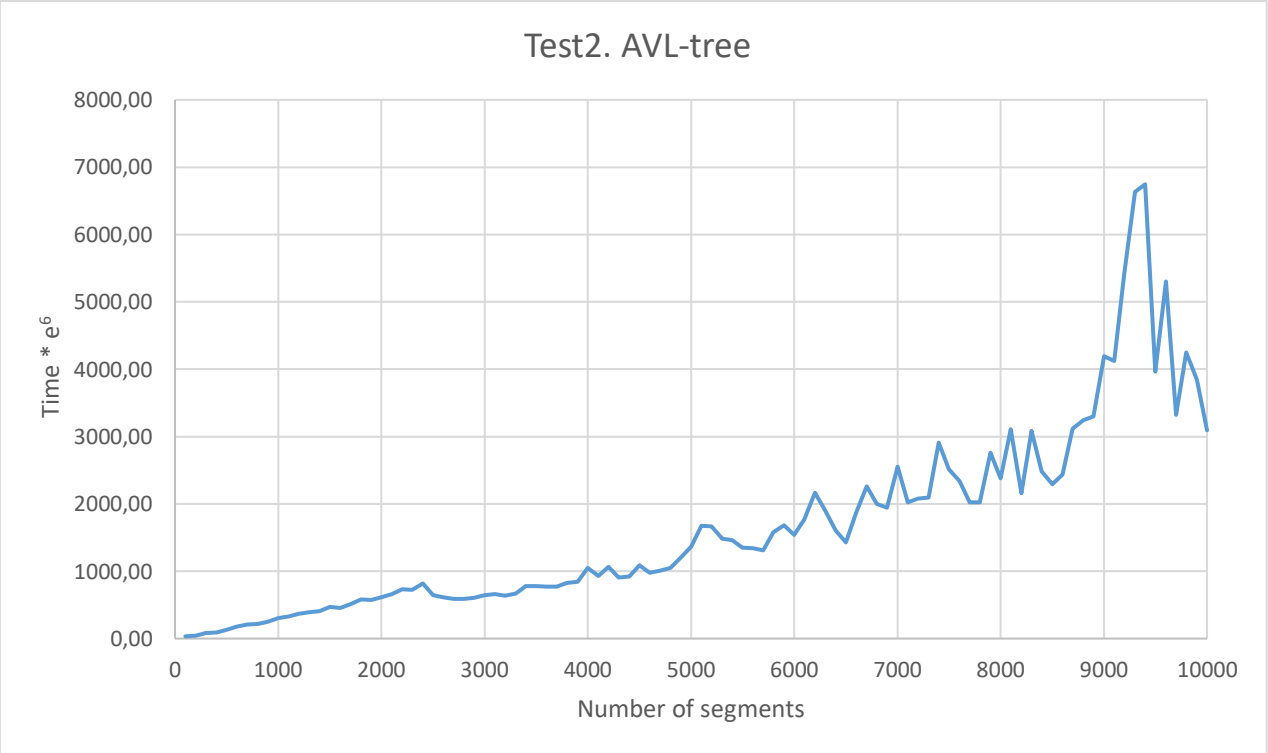


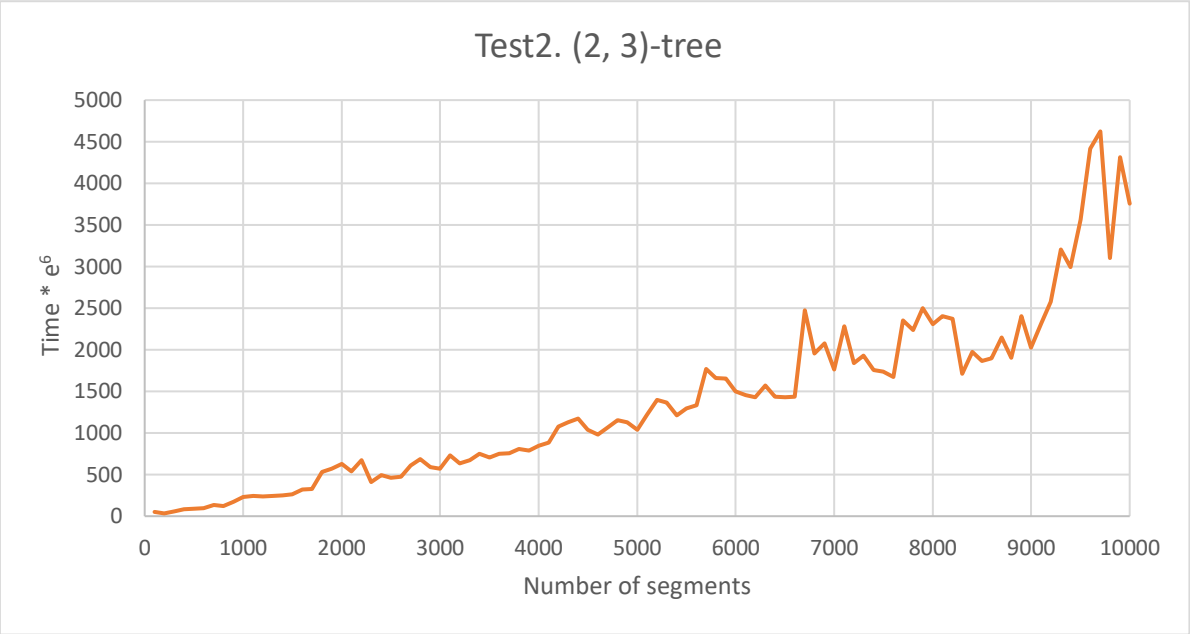


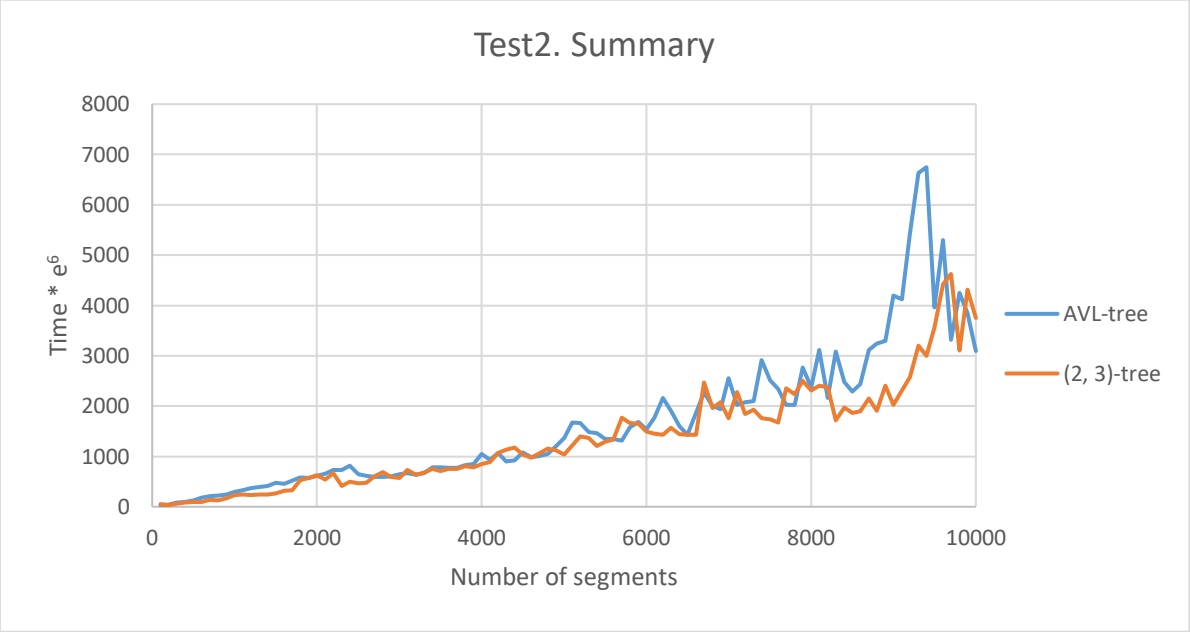
Вывод: для обеих структур данных время выполнения T1(n) показывает схожую логарифмическую зависимость от n благодаря их сбалансированной структуре. Однако (2,3)-дерево в среднем демонстрирует немного меньшее время выполнения из-за меньшего числа перестроек при операциях вставки и удаления, чем в АВЛ-дереве.

Тест 2

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



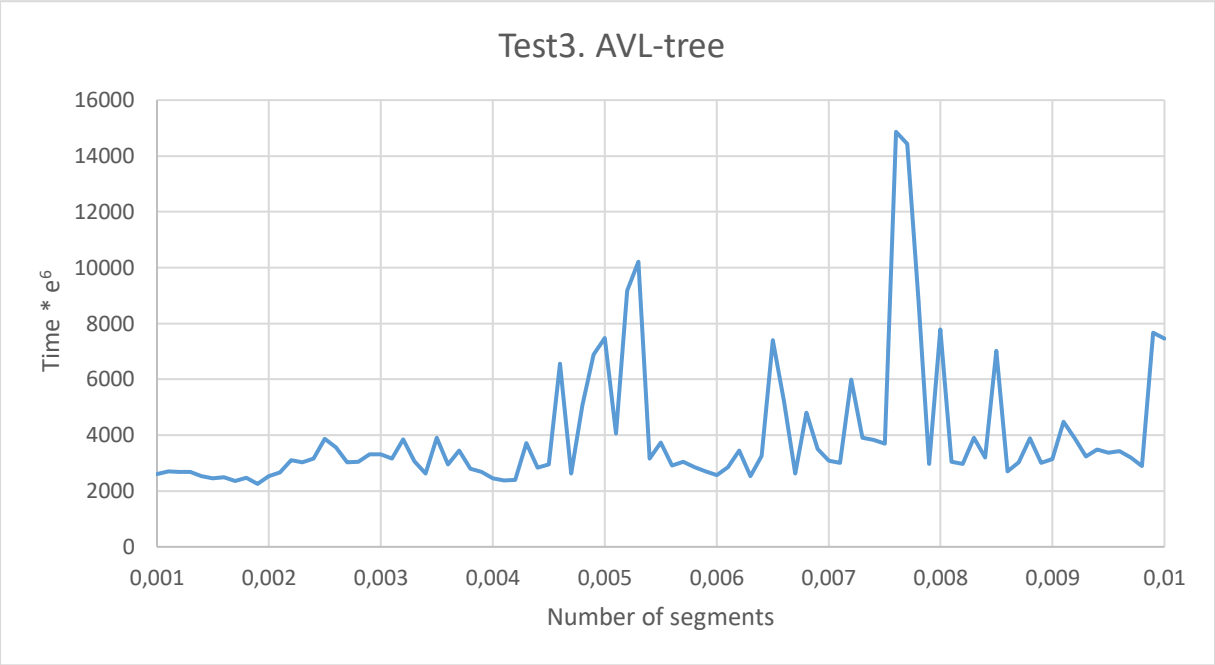


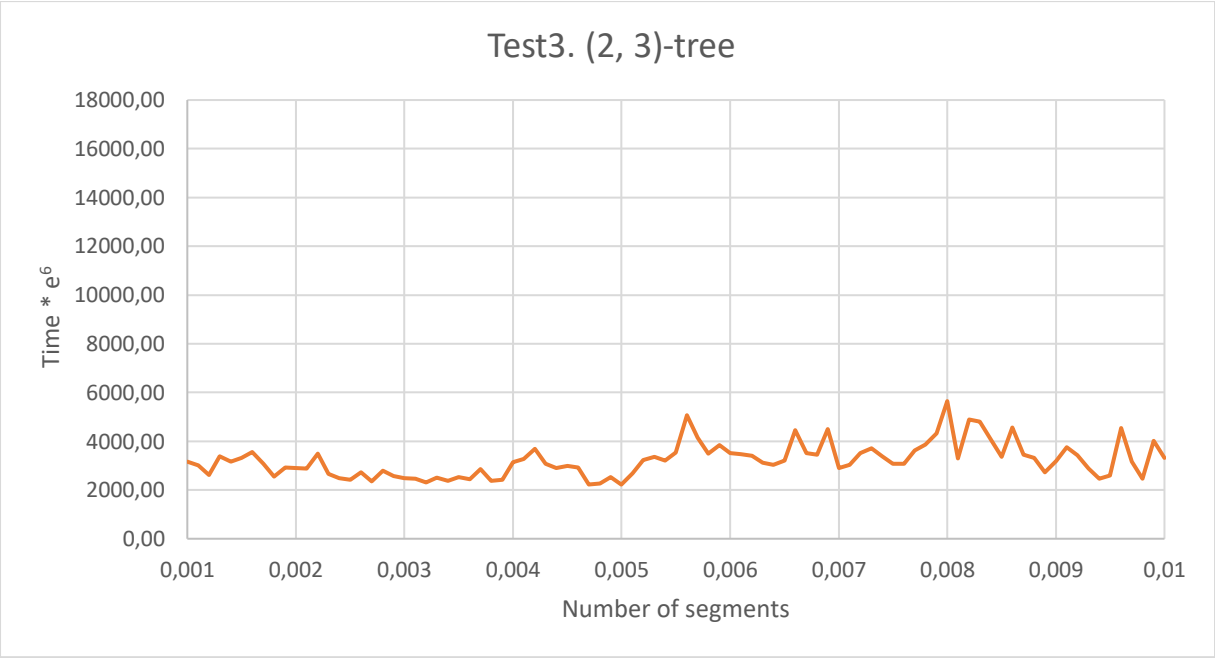


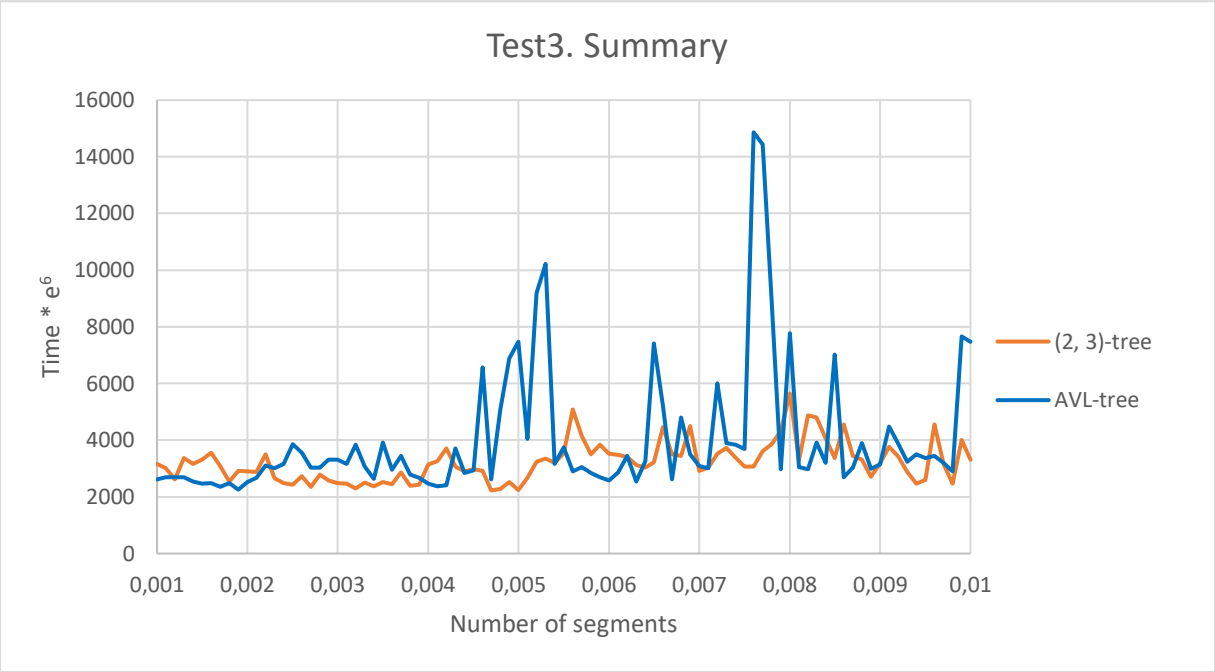
Вывод: сравнение для фиксированного большого n показывает, что рост времени выполнения T1(k) для (2,3)-дерева немного стабильнее, чем для АВЛ-дерева. Это связано с тем, что (2,3)-дерево лучше справляется с последовательным изменением структуры из-за его природы меньшей высоты.

Тест 3

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


Test3. AVL-tree
16000
14000
12000
10000
8000
6000
4000
2000
0
Time * e6
0,001
0,002
0,003
0,004
0,005
0,006
0,007
0,008
0,009
0,01
Number of segments

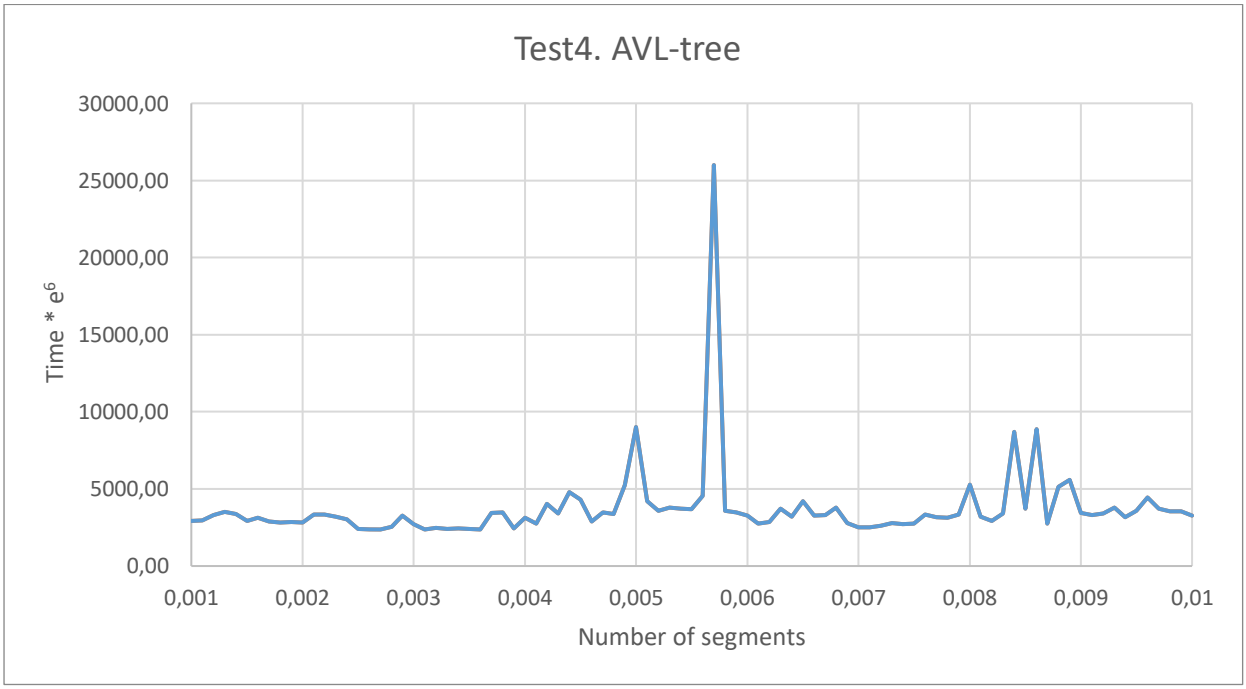

Test3. (2, 3)-tree
18000,00
16000,00
14000,00
12000,00
10000,00
8000,00
6000,00
4000,00
2000,00
0,00
Time * e6
0,001
0,002
0,003
0,004
0,005
0,006
0,007
0,008
0,009
0,01
Number of segments

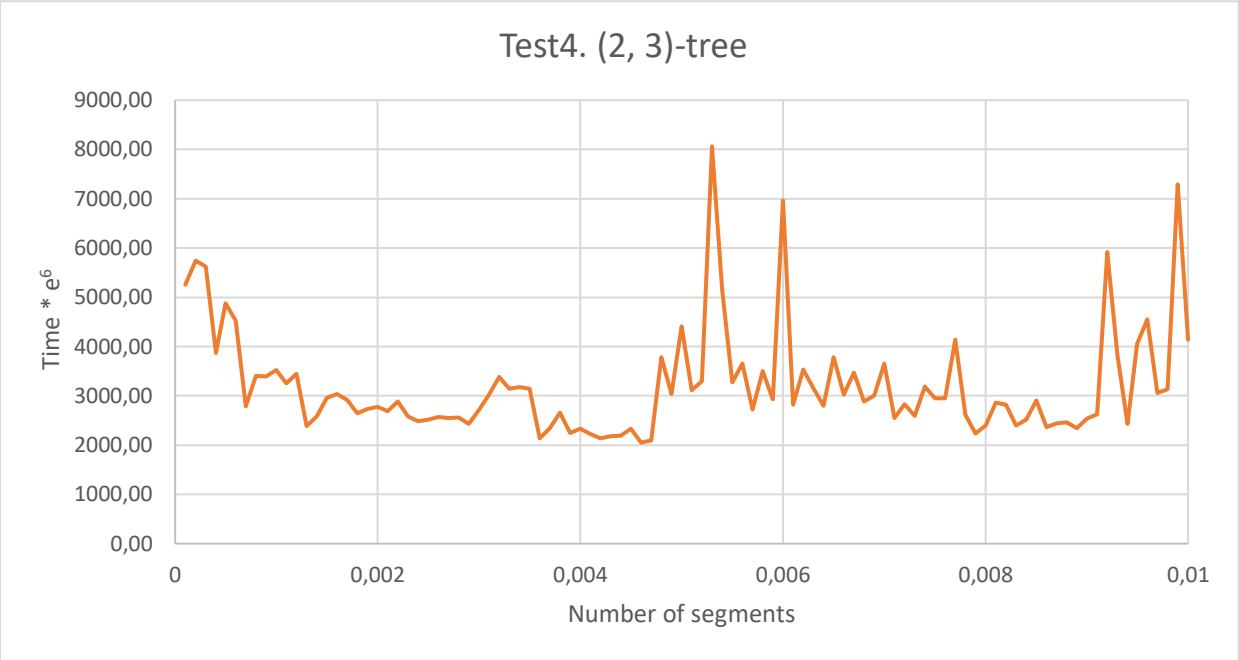


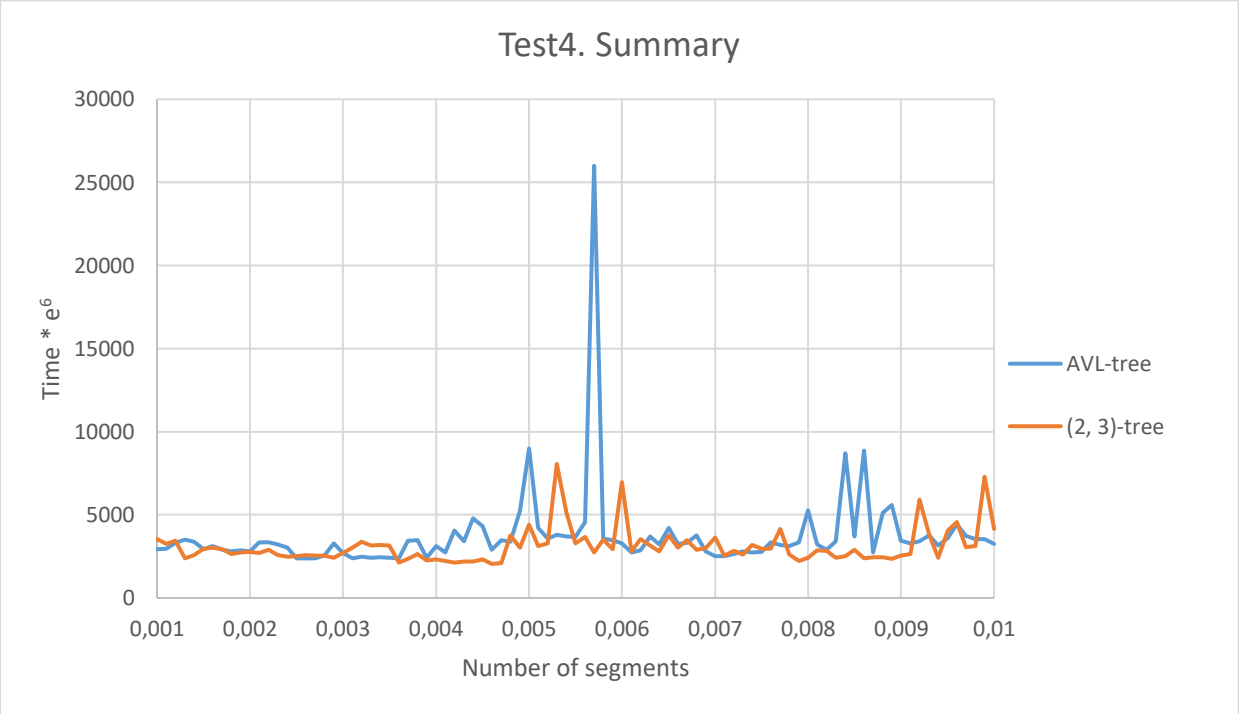
Вывод: при фиксированном r, как АВЛ-дерево, так и (2,3)-дерево показывают хороший рост производительности. (2,3)-дерево имеет небольшое преимущество в скорости операций благодаря меньшей глубине, что снижает среднее время на поиск и модификацию узлов.

Тест 4

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







Вывод: при фиксированном n, зависимость T1(r) показывает, что оба дерева хорошо справляются с разными значениями r. Однако в (2,3)-дереве наблюдается незначительное преимущество из-за меньшего числа перестроек структуры при изменении параметров, что делает его более устойчивым к изменениям.

Общий вывод из экспериментов: сравнение производительности АВЛ-дерева и (2,3)-дерева подтверждает, что обе структуры данных подходят для решения задачи поиска пересекающихся отрезков. Однако (2,3)-дерево, благодаря меньшей высоте и меньшему количеству операций балансировки, демонстрирует небольшое, но стабильное преимущество в скорости выполнения операций по сравнению с АВЛ-деревом, особенно на больших данных.

Заключение

В ходе выполнения работы были исследованы два алгоритма для решения задачи поиска пересекающихся отрезков: с использованием АВЛ-дерева и (2,3)-дерева. Оба подхода реализованы и протестированы на множестве входных данных различной размерности и сложности.

В результате анализа выявлено, что оба алгоритма демонстрируют асимптотическую сложность O(n·log(n)), однако практическое время выполнения зависит от параметров входных данных и особенностей реализации структур данных. АВЛ-дерево обеспечивает высокую производительность операций за счет строго сбалансированной структуры, а (2,3)-дерево демонстрирует устойчивость к изменениям и уменьшает вероятность глубокого дисбаланса.

Список Литературы

1 Ахо А. В., Хопкрофт Д. Э., Ульман Д. Д. Структуры данных и алгоритмы. М.: Мир, 1983. – 400 с.
2 Кормен Т. Х., Лейзерсон Ч. Э., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ. 3-е издание. М.: Вильямс, 2013. – 1328 с.
3 Седжвик Р. Алгоритмы на C++. Часть 3: Структуры данных и алгоритмы. 2-е издание. М.: Вильямс, 2004. – 528 с.