Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Фундаментальные науки
КАФЕДРА Прикладная математика

ОТЧЕТ ПО ПРАКТИКЕ

Студент Пащенко Николай Олегович
фамилия, имя, отчество

Группа ФН2-32Б

Тип практики: Ознакомительная практика

Название предприятия: Научно-учебный комплекс «Фундаментальные науки» МГТУ им. Н.Э. Баумана

Студент _____ Пащенко Н.О.
подпись, дата *фамилия и.о.*

Руководитель практики _____ Попов А.Ю.
подпись, дата *фамилия и.о.*

Оценка _____

2021 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»

(МГТУ им. Н.Э. Баумана)

Кафедра «Прикладная математика»

З А Д А Н И Е
на прохождение ознакомительной практики

на предприятии Научно-учебный комплекс «Фундаментальные науки» МГТУ им. Н.Э. Баумана

Студент Пащенко Николай Олегович
фамилия, имя, отчество

Во время прохождения ознакомительной практики студент должен

1. Изучить на практике основные возможности языка программирования C++ и систем компьютерной алгебры, закрепить знания и умения, полученные в курсах «Введение в информационные технологии», «Информационные технологии профессиональной деятельности».
2. Изучить способы реализации методов решения задачи локализации точки на плоскости.
3. Реализовать метод полос, также называемый «Slab decomposition method».

Дата выдачи задания «20» **сентября 2021 г.**

Руководитель практики _____ Попов А.Ю.
подпись, дата *фамилия и.о.*

Студент _____ Пащенко Н.О.
подпись, дата *фамилия и.о.*

Содержание

Задание

Плоская многоугольная сетка задана следующим образом: дан список точек,каждая из которых задана двумя своими координатами, и список ячеек, для каж-дой из которых записаны последовательно против часовой стрелки номера вер- шин из первого списка точек. При этом гарантируется, что любые две ячейки либо не имеют общих точек, либо имеют одну общую вершину, либо примыкаютк одному общему отрезку (т.е. имеют общую сторону и две общих вершины).

Требуется идентифицировать положение, т.е. установить принадлежность конкретной ячейке, системы точек, про которые известно лишь то, что они лежат строго внутри ячеек.

а) решить задачу «перебором»;

б) решить задачу эффективно, используя «метод полос», называемый также «Slab decomposition method».

Структура исходного файла данных:

n	<< количество узлов сетки
x1 y1	<< координаты первого узла сетки
…	
xn yn	<< координаты n-го узла сетки
p	<< количество ячеек сетки
m1	<< количество углов в первой ячейке
k11 k12 … k1(m1)	<< номера узлов, образующих первую ячейку
…	
mp	<< количество углов в p-й ячейке
kp1 kp2 … kp(mp)	<< номера узлов, образующих p-ю ячейку
q	<< количество точек для идентификации
x1 y1	<< координаты первой точки
…	
xq yq	<< координаты q-й точки

Структура файла результата:

q	<< количество точек для идентификации
c1	<< ячейка, в которую попадает первая точка
…	
cq	<< ячейка, в которую попадает q точка

Введение

Основной целью ознакомительной практики 3-го семестра, входящей в учебный план подготовки бакалавров по направлению 01.03.04 – Прикладная ма-тематика, является знакомство с особенностями осуществления деятельности в рамках выбранного направления подготовки и получение навыков применения теоретических знаний в практической деятельности. В ранее пройденном курсе «Введение в специальность» произошло общее знакомство с возможными направлениями деятельности специалистов в области прикладной математики иполучен опыт оформления работ (реферата), который полезен при оформлении отчета по практике. В рамках освоенного курса «Введение в информационные технологии» изучены основные возможности языка программирования C++ и сформированы базовые умения в области программирования на C++. Задачей практики является закрепление соответствующих знаний и умений и овладение навыками разработки программ на языке C++, реализующих заданные алго- ритмы. Кроме того, практика предполагает формирование умений работы с си- стемами компьютерной алгебры и уяснение различий в принципах построения алгоритмов решения задач при их реализации на языках программирования вы- сокого уровня (к которым относится язык C++) и на языках функционального программирования (реализуемых системами компьютерной алгебры).

1. История рассматриваемой задачи

Локализация точки на плоскости является одной из фундаментальных задач в вычислительной геометрии. Она находит приложения в областях, связанных с обработкой геометрических данных: компьютерная графика , географические информационные системы (ГИС), планирование движения и компьютерное проектирование (САПР). Впервые решить эту задачу, затрачивая $O(\log n)$ времени, удалось Дэвиду П. Добкину и Ричарду Д. Липтону в 1976 году. Однако, необходимая память для этого метода, т.е. хранение необходимых структур данных может достигать $O(n^2)$. Уменьшить объем памяти, необходимый для данного метода удалось Питеру Сарнаку и Роберту Тарьяну, которые стали хранить данные в структуре, называемой persistent red-black tree (персистентное красно-черное дерево). Это позволило им уменьшить память для хранения до размера $O(n)$ [1].

2. Обзор методов решения задачи

2.1 Описание проблемы. Простые случаи

Локализацию точки на плоскости, разделенную на непересекающиеся сегменты можно с уверенностью назвать задачей об определении принадлежности точки многоугольнику. Многоугольник может быть как выпуклым, так и невыпуклым. Обычно предполагается, что многоугольник простой, то есть без самопересечений; но задачу рассматривают и для непростых многоугольников. В последнем случае разные способы определения принадлежности точки многоугольнику могут привести к разным результатам. Различают алгоритмы без предварительной обработки и алгоритмы с предварительной обработкой, в ходе которой создаются некоторые структуры данных, позволяющие в дальнейшем быстрее отвечать на множество запросов о принадлежности разных точек одному и тому же многоугольнику.

Сложность любого алгоритма решения данной задачи измеряется тремя величинами: время запроса, необходимая память, время на предобработку.

Метод 1. Трассировка луча.

Алгоритм данного метода можно описать следующим образом:

1. Выпускаем из искомой точки луч в любом направлении.
2. Считаем количество пересечений с ребрами многоугольника.
3. Если пересечений – четное количество, следовательно точки лежит вне многоугольника; если нечетное – точка лежит внутри многоугольника.

Это основано на том простом наблюдении, что при движении по лучу с каждым пересечением границы точка попеременно оказывается то внутри, то снаружи многоугольника (рис. 1). Алгоритм известен под такими названиями, как crossing number (count) algorithm или even-odd rule. Алгоритм работает за время $O(n)$ для n-угольника.

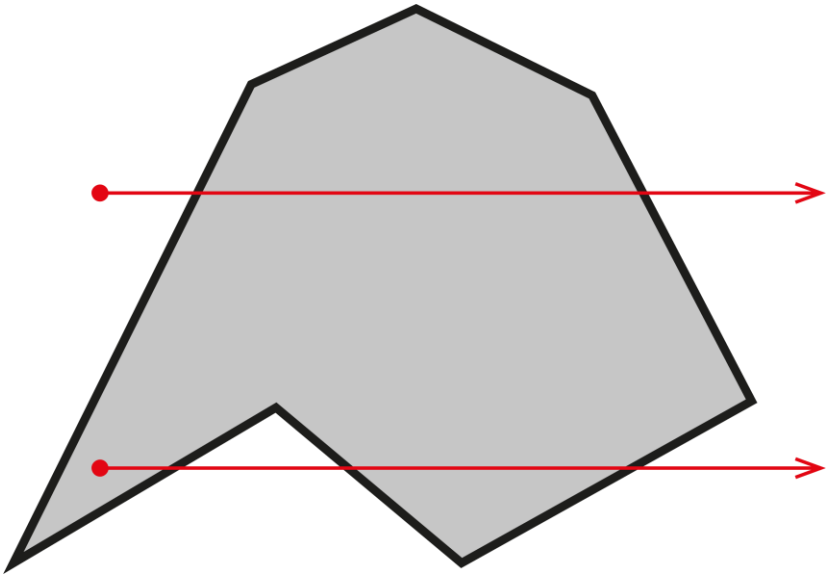

Рис. 1 Метод трассировки луча

Метод 2. Суммирование углов.

Можно определить, что точка P находится внутри многоугольника с вершинами V_0, V_1, ... , $V_n = V_0$, вычислив сумму:

$$\sum_{i=1}^{n} \varphi_i$$

где φ_i - угол (в радианах и со знаком) между лучами PV_{i-1} и PV_i, то есть:

$$\varphi_i = \arccos\left(\frac{PV_{i-1} \cdot PV_i}{|PV_{i-1}| \cdot |PV_i|}\right) sign\left(\det\begin{pmatrix} PV_{i-1} \\ PV_i \end{pmatrix}\right)$$

Можно доказать, что эта сумма есть не что иное, как winding number точки P относительно границы многоугольника, с точностью до константного множителя 2π. Поэтому можно считать, что точка лежит снаружи многоугольника, если сумма равна нулю (или достаточно близка к нему, если используется приближённая арифметика). Однако данный метод весьма непрактичен, так как требует вычисления дорогостоящих операций для каждого ребра (обратных тригонометрических функций, квадратных корней), и был даже назван «худшим в мире алгоритмом» для данной задачи [2].

Кроме того, если система ячеек «склеена», то есть представляет собой планарный граф (такой граф, который можно отобразить на плоскость без пересечений ребер не в вершинах), то имеет смысл решать задачу путем еще большего разбиения поисковой области на более мелкие и более простые для обработки фигуры, такие как треугольники или трапеции. Среди эффективных методов решения задачи можно выделить метод трапеций, метод цепей и метод полос. Из них, с точки зрения затрат на предобработку, наилучшими являются методы цепей и трапеций. В то время как метод полос обладает наихудшим временем на предобработку из всех представленных. А вот с точки зрения затрат времени на запрос наихудшим является метод цепей. Метод трапеций демонстрирует наилучшие результаты и является эффективным методом, но прямая процедура данных поиска использует неоптимальное количество памяти

3.Метод трассировки луча. Описание алгоритма.

Как уже говорилось ранее, существует несколько не самых эффективных, с точки зрения производительности, способов определения местоположения точки на плоскости, а именно в системе ячеек. Один из них – метод трассировки луча.

Полагаем, что из нашей точки (исследуемая точка) выпускается горизонтальный луч, направленный в ту же сторону, что и положительное направление оси абсцисс. После этого нужно проверить количество пересечений лучом

каждой ячейки. Делать это можно, к примеру, в цикле *for()*. Подсчитав количество пересечений луча с ребрами одной ячейки, можно сделать вывод о принадлежности точки ячейке. Если пересечений четное количество – точка не лежит в ячейке, если же нечетное – лежит.

Проверять пересечение ребра ячейки и луча можно также несколькими способами. Например, можно составлять уравнение прямой для каждого ребра, и решать систему из двух уравнений: первое – уравнение прямой соответствующее лучу (при $x > x$ луча) и второе – уравнение соответствующее ребру ячейки. Если данная система будет иметь решение, то засчитываем пересечение. Однако в работе был реализован другой способ определения пересечений. Так как нам известен y точки для поиска, мы знаем y луча (они равны). Также мы знаем x точки, значит все что «слева» от нашей точки нас не интересует. Пользуясь координатами вершин, которые нам также известны, мы можем просто сравнивать координату y начала и конца ребра ячейки. Засчитывать пересечение будем только тогда, когда y начала меньше y луча, а y конца больше y луча, либо же наоборот. Другие ситуации нас интересовать не будут, поэтому в их случае счетчик пересечений не пополняем.

Однако способ осложняется несколькими частными случаями, которые необходимо учитывать.

1. Прохождение луча через вершину ячейки (рис. 2). Для разрешения этой проблемы, необходимо определить координаты y начала текущего ребра и y конца следующего. Если они оба меньше или оба больше чем y луча, то пересечение не засчитываем, а если один из них больше y луча, а другой меньше, то засчитываем пересечение. При этом, после этой ситуации нам нужно «перешагнуть» через одно ребро ячейки, так как оно уже точно не может быть пересечено. То есть рассматривать будем сторону, через одну от текущей.

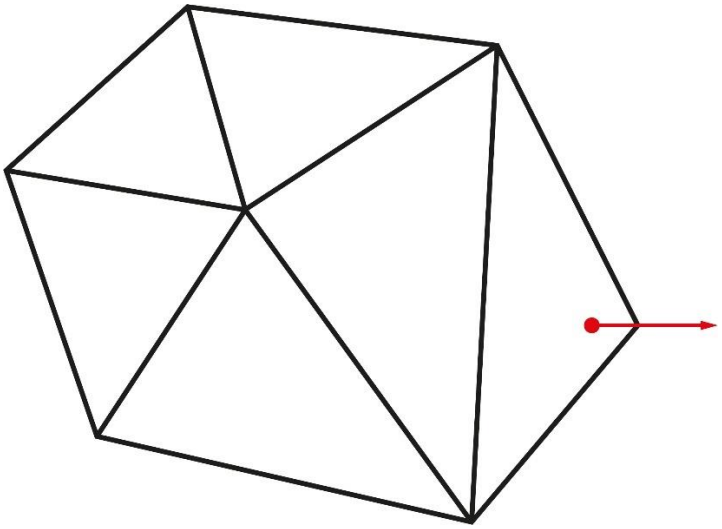

Рис. 2 Прохождение луча через вершину ячейки

2. Совпадение луча со стороной ячейки (рис. 3). При возникновении такой ситуации, нам будет необходимо рассмотреть координату *y* начала предыдущего ребра и координату *y* конца следующего. И по аналогии, если они оба меньше или оба больше координаты *y* луча, то пересечение засчитывать не будем, в противном случае – прибавляем счетчик пересечений.

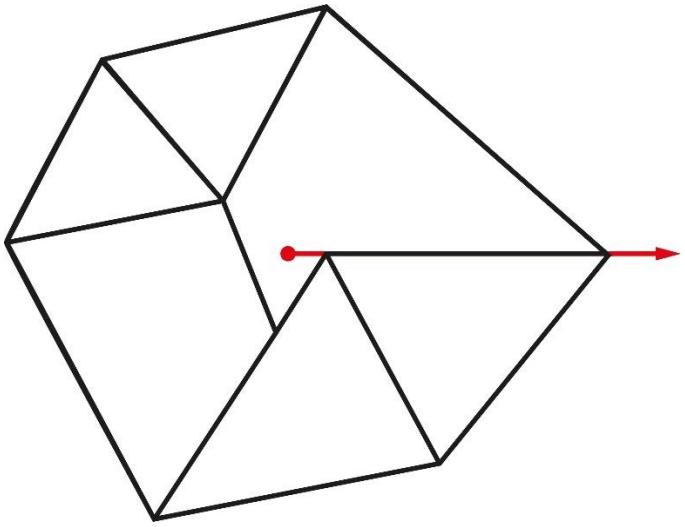

Рис. 3 Совпадение луча со стороной ячейки

Алгоритм метода трассировки луча можно представить следующим образом. Для каждого ребра каждой ячейки:

1. Если ребро лежит левее начала луча, перестаем его рассматривать и переходим к следующему ребру.
2. Если ребро правее начала луча:

2.1. Проверка на случай пересечения луча с вершиной.

2.2. Проверка на случай пересечения луча с ребром (в случае пересечения луча с вершиной).

2.3. Проверка пересечения луча с ребром.

3. В случае пересечения, прибавляем к счетчику пересечений единицу и рассматриваем следующее ребро.

Рассуждая о производительности, конечно, проверка на все частные случаи замедляет работу программы, однако благодаря этому, мы с уверенностью можем определить положение точки даже при нетривиальных структурах ячеек или же при расположении точки, к примеру, на стороне многоугольника. Что касается затрат памяти, для данного способа требуется $O(n)$ памяти для запоминания n точек.

4.1 «Метод полос». Описание алгоритма.

Как мы поняли, эффективными методами решения задачи об определении местоположения точки в системе являются те методы, для которых время запроса составляет $O(\log n)$. Таковым является «метод полос», он же «Slab decomposition method».

Суть данного метода состоит в том, чтобы выполнить два бинарных поиска, сложность каждого из которых $O(\log n)$. Проведя через все вершины ячеек горизонтальные (вертикальные) полосы, мы получим возможность определить точку между двух полос. Это будет первый бинарный поиск. Так как внутри полос ребра ячеек не могут пересекаться, то мы можем упорядочить их слева направо (снизу вверх) и вторым бинарным поиском определить между ребрами какой ячейки лежит наша точка.

Возникает вопрос: а как понять расположение ребер внутри полосы? На помощь приходит такая структура данных как *бинарные деревья*. В случае горизонтальных полос мы можем рассматривать вершины ячеек, лежащие на полосах как *точки событий*. Тогда, предварительно (возьмем за предобработку) отсортировав полосы по координате y снизу вверх, внутри каждой полосы (начиная с первой) мы можем создать дерево упорядоченных слева направо

ребер ячеек. Для каждой точки события в текущее дерево мы будем добавлять или удалять ребра, которые в ней начинаются или заканчиваются. Ключом будет являться координата *x* середины ребра. Внутри полос, содержащих точки для поиска, мы сортируем ребра не по координате *x* середины ребра, а по координате *x* той точки ребра, которая имеет *y*, соответствующий точке для поиска (одинаковый с ее координатой *y*). Таким образом, мы сможем точно определить два ребра, между которыми находится точка.

Алгоритм работы «метода полос»:

1. Проводим через каждую вершину линию и сортируем их по возрастанию.
2. Начиная с первой линии, создаем в каждой полосе бинарное дерево отрезков, исходя из статуса точки события:
   2.1. Если ребер еще не было, то добавляем два ребра.
   2.2. Если было одно ребро, то удаляем его и добавляем другое.
   2.3. Если было два ребра, то удаляем их, и ничего не добавляем.
3. Дойдя до полосы с точкой, которую нужно идентифицировать, создаем в ней дерево со специальным ключом, и определяем между ребрами какой ячейки она лежит.

Что касается памяти, то ее для реализации данного способа в общем случае нужно $O(n^2)$, где n – количество вершин всех ячеек. Дело в том, что в худшем случае, у нас будет n различных полос, в которых будут последовательно находится все ребра всех ячеек. Однако существуют некоторые способы, которые позволяют этого избежать, один из них – это использование частично персистентных деревьев. Такие деревья позволяют хранить все свои версии изменений не в качестве копий дерева, а в качестве скрытых указателей, которые обозначают, какой была та или иная версия [3]. И в таком случае памяти потребуется $O(n)$ – просто для запоминания всех вершин.

Интересно то, что создание деревьев так же относится к предобработке, и на время запроса оно влиять не будет, однако, для конкретной системы ячеек

первый прогон программы будет несколько длиннее последующих, так как будут создаваться и запоминаться *m* бинарных деревьев, где *m* – количество полос. Тогда можем определить количество времени, необходимое для предобработки: $O(n \cdot \log n)$ – для создания и запоминания системы деревьев, $O(n \cdot \log n)$ – для сортировки полос. Это оптимальный вариант, если работа предстоит с одной и той же системой ячеек некоторое количество раз. Учитывать сортировку точек для поиска по возрастанию их координаты *y* стоит лишь тогда, когда их количество соизмеримо, к примеру, с количеством точек, образующих многоугольную сетку. В большинстве случаев это не так, и точек для поиска на порядок меньше.

5. Метод перебора. Особенности реализации

5.1. Реализация на C++

Нами было принято решение использовать в программе на языке программирования C++ объектно-ориентированный подход, и в большей степени одну из его парадигм – инкапсуляцию. Мы реализовали три класса: класс точки – *Point,* класс отрезка – *Segment,* класс ячейки – *Cell*. Эти классы имеют закрытые данные-члены и открытые методы, которые позволяют нам безопасно работать с самими данными. Некоторые классы реализованы посредством других классов. Например, *Segment* содержит два поля данных типа *Point*. Поэтому между ними установлены «дружеские» отношения, а именно класс *Segment* объявлен другом для класса *Point*. Таким образом он сможет иметь доступ к закрытым данным-членам *Point*.

Самыми важными функциями являются *Segment::travers* и *Cell::spot*. Первая проверяет, может ли луч в принципе пересечь ребро. Она исключает все ребра, которые по тем или иным причинам не могут быть пересечены (например, те, что находятся левее начала луча). Вторая функция выполняет основную работу по инициализации точки. Именно в ней в большей степени реализован вышеописанный алгоритм метода трассировки луча. Объект класса *Cell* вызывает метод *spot* и передает ему точку, местоположение которой нужно найти. Результат – значение номера ячейки.

Кроме этих функций, было реализовано много вспомогательных функций. Одна из них – функция *line* – проверяет совпадение луча и ребра ячейки. Также, для объектов класса *Point* и *Cell* были переопределены некоторые операторы: для *Point* – *Point::operator<,* для *Cell* – *Cell::operator[].* Класс *Cell* имеет поле, которое является указателем на динамически выделенный массив. Деструктор класса освобождает эти данные. Это было сделано для более рационального использования памяти.

Касаемо памяти, было создано три контейнера типа *std::vector.* Один из них хранит точки (объекты класс *Point*), образующие плоскую сетку, другой хранит объекты типа *Cell,* и третий хранит точки для идентификации, которые также были представлены как объекты класса *Point.*

Таким образом, при реализации метода перебора было сделано некоторое заключение. Успех написания программы во многом зависит от правильного и удобного представления данных, для той или иной задачи. Следовательно, при знании некоторых техник, правил и стилей написания кода, сложность разработки может заметно снизиться.

5.2. Реализация в среде Wolfram Mathematica

Система компьютерной алгебры Wolfram Mathematica предоставляет широкий диапазон возможностей для решения тех или иных математических и прикладных задач. Задача, которую предстояло нам решить не была исключением. Для представления данных, мы использовали четыре списка, каждый из которых выполнял свою задачу. Первый список содержит в себе координаты точек, образующих плоскую многоугольную сетку. Второй список содержит номера точек, образующих ячейки. Третий список содержит координаты вершин ячеек. Четвертый список содержит точки для поиска.

Заполнив в цикле все эти списки данными, мы прошлись по всем ячейкам и, применив функцию *RegionMember,* определили, лежит ли точка внутри конкретной ячейки или нет.

Для визуализации мы использовали функцию *ListLinePlot* и некоторые ее

параметры. Для чтения из файла и записи в него были применены такие команды, как *Input* и *Output*.

6. «Метод полос». Особенности реализации

6.1. Реализация на C++

Перед написанием кода необходимо было тщательно продумать структуру программы, дабы избежать лишних трат памяти, так как и без того способ является достаточно ресурсозатратным. Было выбрано решение наиболее часто использовать ссылочный тип данных, а также указатели.

В программе также реализованы классы, но уже не три, а четыре. Класс *Point* теперь имеет наследника – класс *Point_for_cell,* который является вершиной ячейки. Это было сделано потому, что объект класса *Point_for_cell* будет выполнять роль точки события, а она должна хранить некоторую информацию об отрезках, которые из нее выходят. Так и есть, *Point_for_cell* содержит два члена-данных типа *Segment**. Они и есть те два отрезка, которые выходят из точки события.

Cell стал своего рода «вложенным» типом данных. Он как и прежде содержит указатель на массив вершин, реализованных классом *Point_for_cell*, каждая из которых хранит указатель на два ребра – левое и правое, которые в себе содержат номер ячейки *Segment::num*, и коэффициенты *Segment::k* и *Segment::b*, определяющие прямую на плоскости. Однако это не повлекло критических потерь в памяти, так как внутри себя объекты используют указатели на одни и те же места в памяти. К примеру, две последовательные вершины ячейки, имеют в себе указатель на одно и то же ребро (рис. 4).

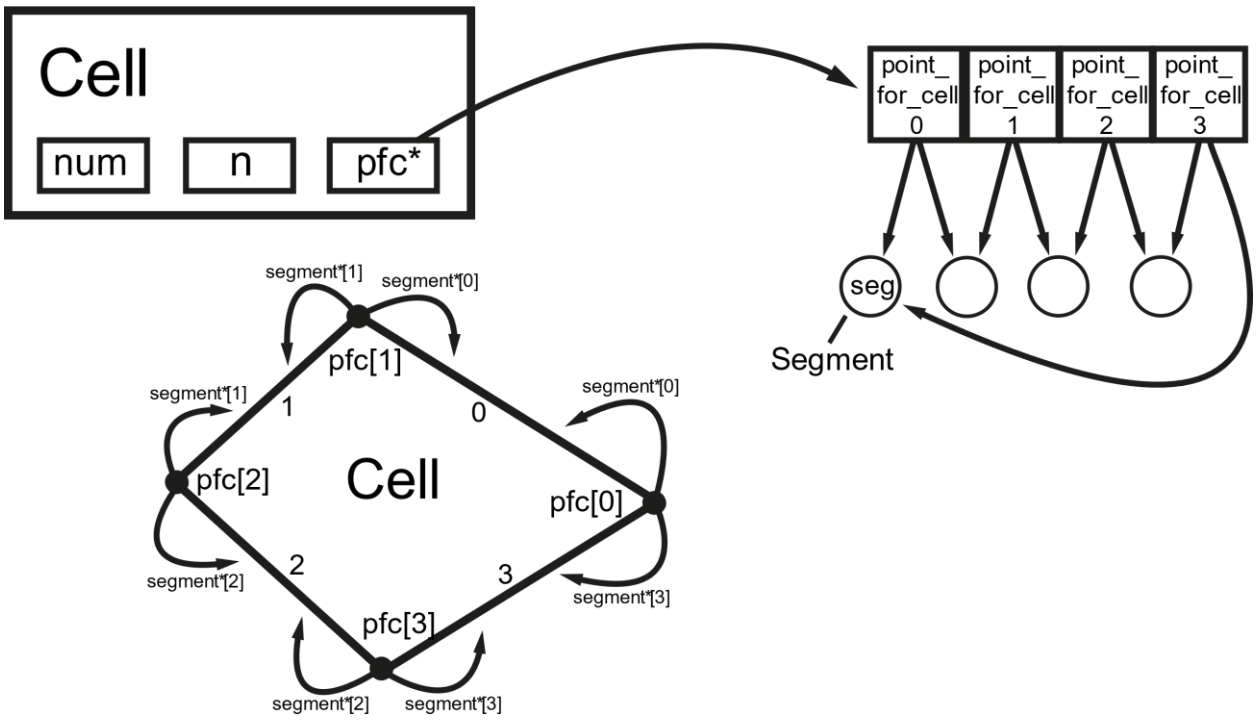

Рис. 4 Схема хранения данных в реализации сложного метода

Для хранения точек и ячеек, как и раньше, использованы два контейнера *std::vector<Point>* и *std::vector<Cell>*. Также нам потребовались две структуры для хранения полос, и для хранения точек событий. Для этого были использованы контейнеры *std::map<double, Point*>* и *std::multimap<double, Point_for_cell*>* с ключом – координатой *y*. Их преимущество заключается в том, что они реализованы посредством красно-черного дерева, и поэтому вставка, поиск и удаление происходят за постоянное время *O(log n)*, где *n* – количество элементов в контейнере [4]. Таким образом, после считывания данных из файла и записи их в вышеуказанные контейнеры, мы имеем отсортированный по возрастанию координаты *y* список полос и список точек событий.

Найдя все полосы, в которых содержатся точки для поиска, мы начинаем построение последовательностей отрезков на каждой полосе. Для этого используем контейнер вида std::vector<std::multimap<double, segment*>>. Размером этого контейнера является максимальный номер полосы, в которой есть точка для поиска (дальше деревья на полосах можно не строить, так как выше не лежит ни одной искомой точки).

Далее мы начинаем последовательное построение деревьев на каждой полосе. Ключом, как и сказано в алгоритме, будет являться координата *x* середины ребра. Одними из наиболее часто используемых функций стали методы контейнеров *std::map* и *std::multimap*, такие как *std::emplace* – вставка, *std::erase* – удаление по ключу, *std::find* – поиск по ключу, *std::equal_range* – поиск диапазона элементов с заданным ключом. При попадании на полосу с нужным нам номером (где лежит точка для поиска) мы пересортировываем дерево по новому ключу – координате *x*, соответствующей *y* точки и вставкой в это дерево определяем, между какими двумя ребрами находится точка. Записываем номер ячейки и движемся дальше.

В программе пришлось изменить порядок следования точек для поиска, чтобы удобно расположить их по возрастанию координаты *y*. Для того, чтобы затем вернуть этот порядок, был создан вспомогательный контейнер типа *std::vector*, но как говорилось в описании алгоритма, часто точек для поиска не так много, и эта процедура не займет очень много времени и ресурсов.

Таким образом, как и предполагалось, при реализации данного метода пришлось жертвовать памятью и временем предобработки ради улучшения времени запроса.

7. Результаты тестовых примеров

Пример 1.

Исходные данные:

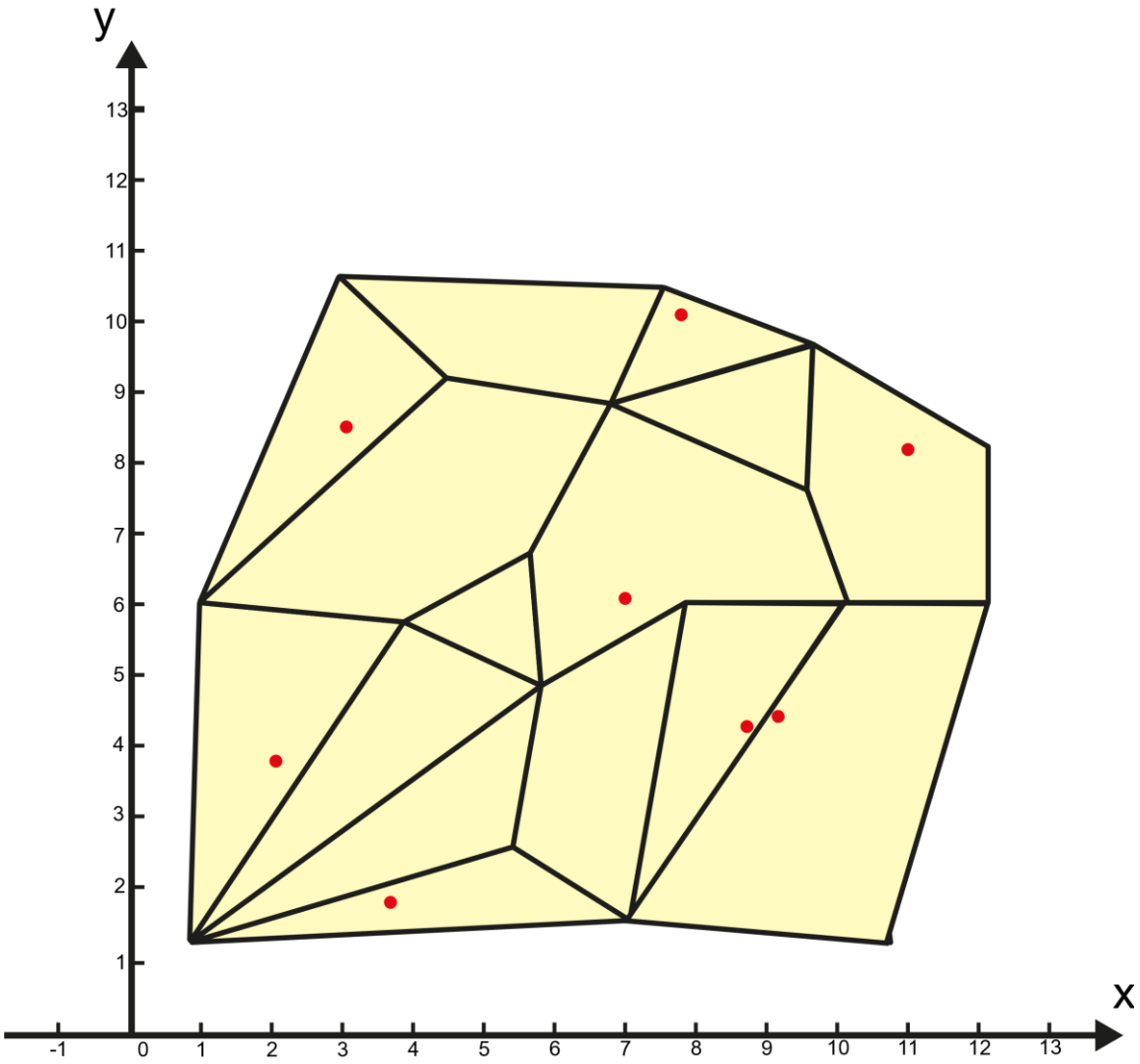

Рис. 5 Сетка для примера № 1

Результаты метода трассировки луча на C++: за 0.001 секунды находит верно все точки и записывает в файл верный результат.

Результаты метода полос на C++: за 0.001 секунды находит верно все точки и записывает в файл верный результат.

Результаты метода перебора в Wolfram Mathematica: за 0.168 секунд находит верно все точки и записывает в файл верный результат.

Пример 2.

Исходные данные:

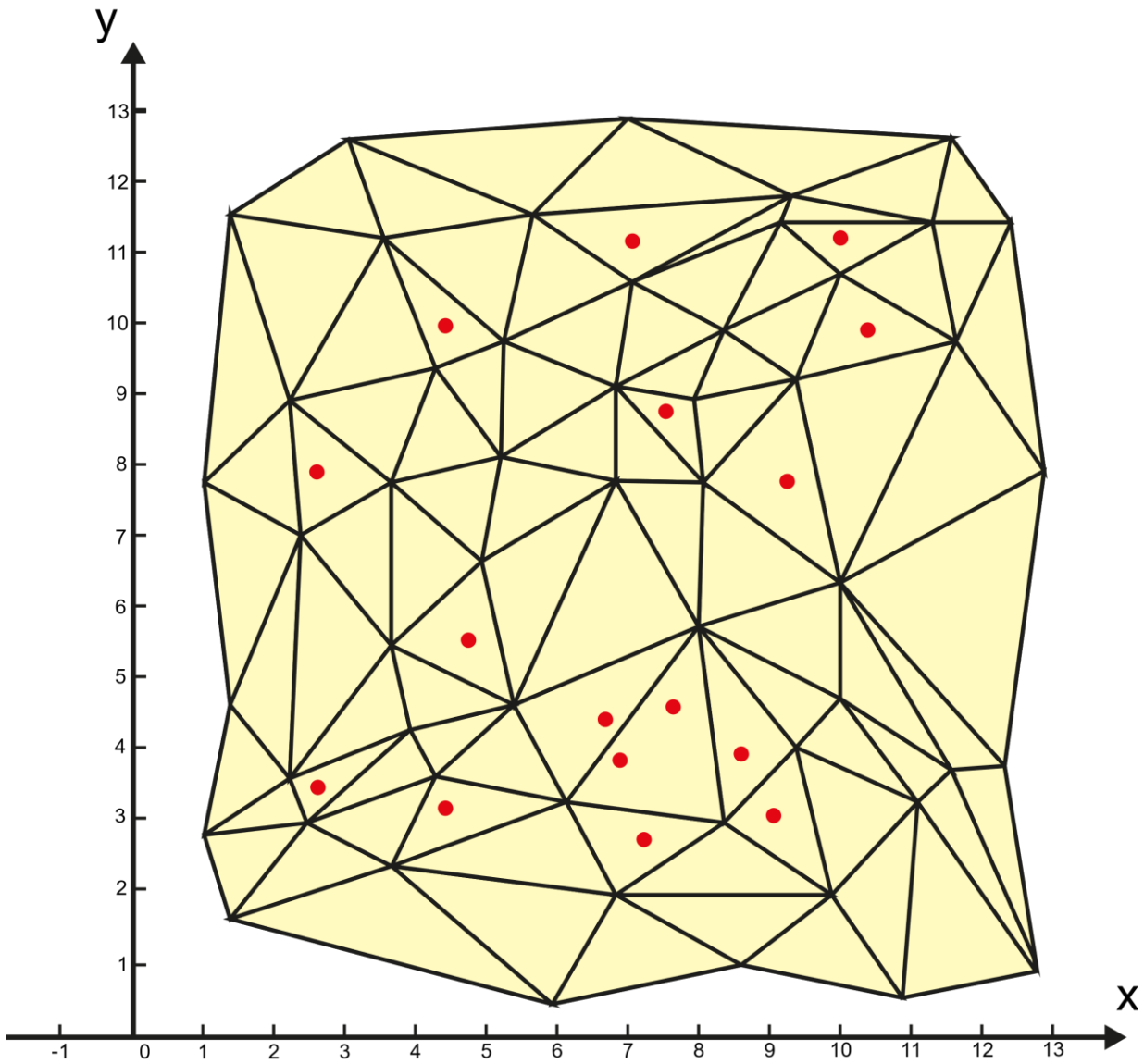

Рис. 6 Сетка для примера № 2

Результаты метода трассировки луча на C++: за 0.002 секунды находит верно все точки и записывает в файл верный результат.

Результаты метода полос на C++: за 0.0015 секунды находит верно все точки и записывает в файл верный результат.

Результаты метода перебора в Wolfram Mathematica: за 0.35 секунд находит верно все точки и записывает в файл верный результат.

Заключение

В процессе решения задачи локализации точки на плоскости в формате практики 3 семестра обучения мы познакомились с несколькими методами решения данной задачи и изучили некоторые из них. Нам удалось закрепить полученные в курсе «Введение в информационные технологии» знания и умения программирования на языке C++. Более того, мы узнали много нового, например, нам пришлось разбираться с такой структурой хранения данных, как персистентные красно-черные деревья. Также, мы закрепили знания по курсу «Введение в специальность» путем реализации программы в системе компьютерной алгебры Wolfram Mathematica.

Список литературы

1. Расположение точки // Википедия. Свободная энциклопедия. URL: https://en.wikipedia.org/wiki/Point_location
(дата обращения: 08.11.21)

3. Задача о принадлежности точки многоугольнику // Википедия. Свободная энциклопедия. URL: https://ru.wikipedia.org/wiki/Задача_о_принадлежности_точки_многоугольнику
(дата обращения: 05.10.21)

3. Красно-черное дерево // Википедия. Свободная энциклопедия. URL: https://ru.wikipedia.org/wiki/Красно_черное_дерево
(дата обращения 15.11.21)

4. std::map // Cppreference. Справочник C++. URL: https://en.cppreference.com/w/cpp/container/map
(дата обращения: 15.11.21).