Лабораторная работа №2 (Канделов Дамир 22Пи-1)

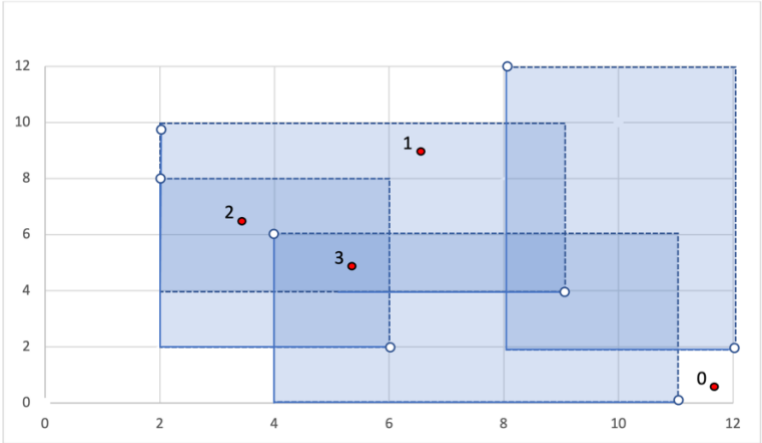
Скольким прямоугольникам на плоскости принадлежит точка?

Описание:

В данной лабораторной работе мы рассматривали различные алогоритмы, которые отвечают на вопрос: Скольким прямоугольникам на плоскости принадлежит точка?

Задача:

Задание данной лабораторной работы заключалось в реализации и сравнении трех алгоритмов: алгоритм перебора (BruteForce), алгоритм на карте (MapAlgorithm), алгоритм на дереве отрезков (PersistentSTAlgorithm).



1. **Алгоритм перебора (BruteForce)**

   Первый алгоритм самый простой - метод полного перебора. Для каждой пришедшей на вход точки мы проверяем все прямоугольники: лежит точка внутри или нет.

   Сложность подготовки: **O(1)**, ее тут нет.
   Сложность поиска: **O(N)**, где N - количество прямогольников.
   Общее время работы: **O(N * M)**, где N - количество прямогольников, а M - количество точек.

   Сложность по памяти: **O(1)** – дополнительная память не требуется.

2. **Алгоритм на карте (MapAlgorithm)**

   В этом алгоритме мы сначала сжимаем координаты, затем строим карту. Потом при помощи бинарного поиска отвечаем на вопрос для каждой точки.

   Сложность подготовки: **$O(N^3)$** - мы идем по всем прямоугольникам, затем для каждого проходимся по всем точкам нашей карты, которые принадлежат прямоугольнику, это $O(N^2)$, так как мы сжали карту по координатам и размер нашей карты примерно N * N.
   Сложность поиска: **O(logN)** – используем бинарный поиск для того чтобы найти нужную ячейку на карте сначала по X, затем по Y.
   Общее время работы: **$O(N^3 + M * logN)$.**

   Сложность по памяти: **$O(N^2)$** – мы храним в памяти карту примерно N * N.

3. **Алгоритм на дереве (PersistentSTAlgorithm)**

   Самый интересный алгоритм: сочетает в себе scanline и персистентное ДО.

   То есть: сортируем прямоугольники по их координатам по X, затем идем по ним, когда доходим до начала прямоугольника – в дереве отрезков по сжатым Y координатам добавляем единицу на отрезке от начала прямоугольник до конца по Y. Когда встречаем конец прямоугольника – делаем тоже самое, но вычитаем единицу, а не прибавляем ее. Если начала и конца прямоугольников в этой координате X закончились, то сохраняем текущее состояние дерева и переходим к следующей координате X, и так до конца.

Сложность подготовки: **O(N * logN)** – мы строим дерево + идем по всем прямоугольникам + на каждом шаге меняем значения в дереве (O(logN)).
Сложность поиска: **O(logN)** – используем бинарный поиск для того чтобы найти нужное состояние дерева, затем в дереве находим ответ. Операции выполняем одну за другой => O(logN).
Общее время работы: **O(N * logN).**

Сложность по памяти: **O(N + N * logN)** – посчитано очень грубо: изначальное дерево отрезков нам нужно хранить => это примерно O(N) памяти, далее мы на каждом шаге, всего их N, изменяем дерево (примерно одну его ветвь), то есть грубо говоря добавляется logN новых вершин => O(logN * N). Это работает только если те ветви, которые мы не трогали брать из предыдущих состояний.

	Сложность подготовки	Сложность поиска	Общая сложность	Сложность по памяти	Ссылка на посылку в контесте (может быть откроется с админ правами)
Алгоритм перебора	**O(1)**	**O(N)**	**O(N * M)**	**O(1)**	https://contest.yandex.ru/contest/60324/run-report/112983576/
Алгоритм на карте	**$O(N^3)$**	**O(logN)**	**$O(N^3 + M * logN)$**	**$O(N^2)$**	https://contest.yandex.ru/contest/60324/run-report/112997628/
Алгоритм на ДО	**O(N * logN)**	**O(logN)**	**O(N * logN)**	**O(N + N * logN)**	https://contest.yandex.ru/contest/60324/run-report/113215908/

Результаты:

В ходе лабораторной работы были реализованы все алгоритмы, написаны юнит-тесты для них, а также все алгоритмы были протестированы в контесте.

Репозиторий с кодом:
https://github.com/Kaparya/HSE/tree/main/Courses/Algorithms%20and%20data%20structures/Lab2%20(points%20in%20rectangles)

Реализации всех алгоритмов лежат в папке src/Algorithms/
В проекте присутствует три режима работы: TESTING (тестирование на юнит-тестах, падение программы в случае не прохождения), TIME_SCORING (замер времени и вывод в файлы), MANUAL_INPUT (ручной ввод). Чтобы изменить режим требуется раскомментить соответствующую строку.

```
4  //#define MANUAL_INPUT
5  //#define TESTING
6  #define TIME_SCORING
```

В данный момент включен режим замера времени.

В папке Description/Results лежат все сырые результаты, которые использовались для построения графиков. Последний график строился в питоне (файл src/graph_builder.py).
Тестовые данные лежат в tests/tests.cpp, для замера времени использовался измененный один из юнит-тестов BigTest.

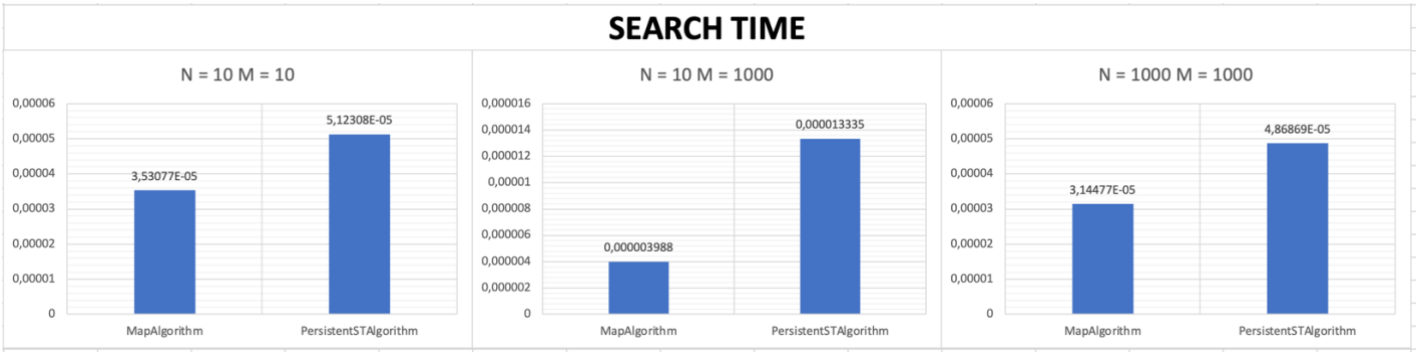
Почта в Яндекс.Контесте: drkandelov@edu.hse.ru (или будет отображаться Канделов Дамир).

Далее были протестированы алгоритмы на разных наборах данных:

Наверное, есть смысл сравнивать алгоритмы только по общему времени работы, так как если сравнивать их, например, по времени подготовки, то у BruteForce – это O(1), то есть практически нуль. У MapAlgorithm – $O(N^3)$, а у PersistentSTAlgorithm – O(N * logN). Соответственно понятно, что у MapAlgorithm самое плохое время, а у BruteForce самое хорошее.

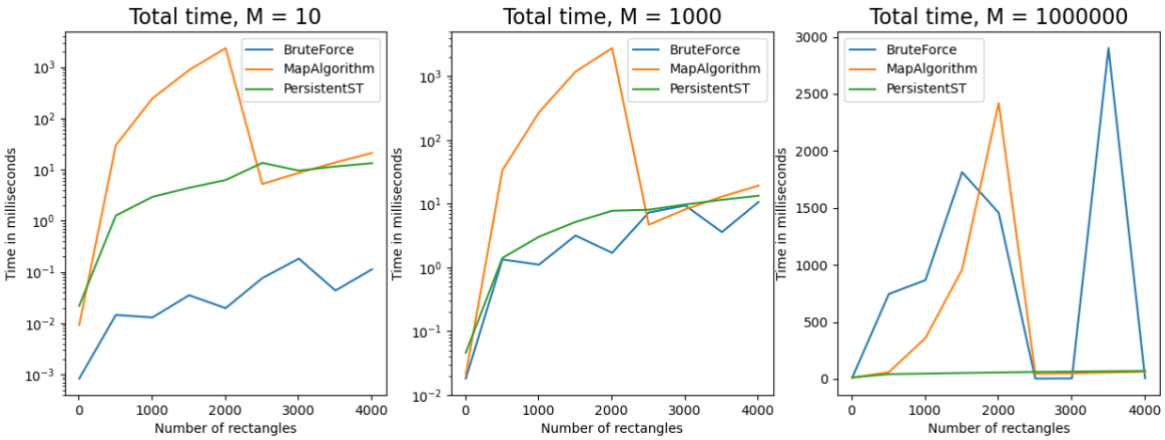
Однако сделаем еще одно сравнение – по времени поиска: заметим, что сложность поиска у алгоритма на карте и на дереве одинакова и равна O(logN). (Сложность поиска у BruteForce – O(N) => даже не будем включать его в сравнение, это гораздо больше.)
Как мы знаем, в таком случае время все равно может отличаться и зависеть от константы, на которую умножается эта сложность. Соответственно, тут мы видим, что всегда оказывается меньше время поиска у MapAlgorithm => константа в этом алгоритме меньше. Это понятно, так как тут мы просто делаем бинпоиск по координатам сначала X, потом Y и получаем ответ просто глядя в нашу таблицу

по получившимся индексам. А в алгоритме на дереве мы тоже начинаем с бинпоиска по X, однако потом нам надо пройтись от корня дерева до листа. Дерево реализовано на указателях, а значит оно может хранится не в соседних ячейках памяти => при каждом переходе мы достаем из памяти в кэш разные кусочки памяти, что требует дополнительного времени. Вот тут мы и получаем константу больше, чем в первом случае. (Но важно отметить, что тут очень маленькие числа времени и скорей всего большая погрешность).



И самое интересное сравнение – сравнение общего времени.
Как мы видим, в случаях, когда N маленькое или среднее и M маленькое или среднее - у нас побеждает алгоритм BruteForce. Это очень важное наблюдение, так как если данные будут именно такими, то BruteForce оптимально использовать не только из-за хороших показателей времени, но и потому что его очень просто написать и допустить ошибку почти невозможно.
Однако, при большом количестве точек от пользователя BruteForce начинает очень сильно проигрывать другим алгоритмам. Так, при M = 10^6 и маленьком N – Алгоритмы на карте и на дереве очень близки по времени, поэтому стоит задуматься, что использовать: MapAlgorithm гораздо проще реализовать и отследить ошибки.
При больших значениях обоих параметров уже побеждает PersistentSTAlgorithm – все-таки у него самая лучшая сложность.

Проанализируем также графики чуть-чуть сложнее:
Тут мы видим, что при M = 10 и 1000 наши догадки подтвердились, BruteForce показывает самое хорошее время. Тут прекрасно видно, что время работы MapAlgorithm очень сильно растет, а потом значения начинают буферизироваться и уже не растут дальше так сильно. Однако при запуске не подряд значения бы продолжили расти. Алгоритм на дереве тоже показывает себя не очень хорошо: при маленьких M большое время подготовки проигрывает обычному BruteForce.
По третьему графику же уже гораздо лучше видно, почему при больших входных данных используются такие сложные алгоритмы, как PersistentSTAlgorithm: его время работы остается очень маленьким даже при большом количестве точек. (тут присутствуют очень большие погрешности при подсчете времени работы у BruteForce и у MapAlgorithm: значения буферезируются, также компилятор оптимизирует алгоритмы и тд, но все же видно, что PersistentSTAlgorithm лучше других двух).



Вывод:

Мы реализовали и сравнили 3 алгоритма, которые помогают понять, скольким прямоугольникам принадлежит точка на плоскости, а также заметили, что алгоритмическая сложность алгоритмов – это важно, но также нужно помнить и про константу, на которую умножается эта сложность при работе алгоритма. Невозможно точно сказать, какой алгоритм лучше: каждый из них хорош при тех или иных условиях. Плюсы и минусы всех трех алгоритмов:

BruteForce:
+ Легкий в плане написания кода
+ Оптимален для небольших значений N и M
+ Не требует дополнительную память
+ Можно добавлять в процессе работы как новые прямоугольники, так и новые точки
- Очень сильно проигрывает по времени при увеличении N или M

MapAlgorithm:
+ Константа меньше, чем в алгоритме на дереве. То есть, если, например, заранее проводить этап подготовки (где у этого алгоритма сложность $O(N^3)$), то лучше использовать этот алгоритм для больших значений M
+ Несложный в плане написания кода
- Нельзя добавлять в процессе работы новые прямоугольники
- Требуется много дополнительной памяти $O(N^2)$
- Слишком большая алгоритмическая сложность этапа подготовки

PersistentSTAlgorithm:
+ Хорошее время работы на больших значениях N и M
+ Требуется немного дополнительной памяти O(NlogN)
- Нельзя добавлять в процессе работы новые прямоугольники
- Сложный в плане написания и отладки
- Требуется следить за очищением памяти. Если это «сырые» указатели, то не совсем понятно, как их потом удалять, решение - умные указатели (у меня std::shared_ptr).