Московский авиационный институт (национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Студент: С. Ю. Свиридов
Группа: М8О-306Б-22
Дата:
Оценка:
Подпись:

Москва 2025

Курсовой проект 24/25

Задача: I. Персистентные структуры данных

Вам дан набор горизонтальных отрезков, и набор точек. Для каждой точки определите сколько отрезков лежит строго над ней. Ваше решение должно работать online, то есть должно обрабатывать запросы по одному после построения необходимой структуры данных по входным данным. Чтение входных данных и запросов вместе и построение по ним общей структуры запрещено.

Формат ввода

В первой строке вам даны два числа n и $m(1 \leq n, m \leq 10^5)$ - количество отрезков и количество точек соответственно. В следующих n строках вам заданы тройки чисел l, r и $h(-10^{-9} \leq l < r \leq 10^9, -10^{-9} \leq h \leq 10^9)$ - координаты x левой и правой границ отрезка и координата y отрезка соответственно. В следующих m строках вам даны пары чисел $x, y(-10^9 \leq x, y \leq 10^9)$ - координаты точек.

Формат вывода

Для каждой точки выведите количество отрезков строго над ней.

1 Описание и идея решения

Персистентные структуры данных - это структуры данных, которые при внесении в них каких-то изменений сохраняют все свои предыдущие состояния и доступ к этим состояниям. Существует несколько уровней персистентности:

1. Частичная. В частично персистентных структурах данных к каждой версии можно делать запросы, но изменять можно только последнюю версию структуры данных.

2. Полная. В полностью персистентных структурах данных можно менять не только последнюю, но и любую версию структур данных, также к любой версии можно делать запросы.

3. Конфлюэнтная. Конфлюэнтные структуры данных позволяют объединять две структуры данных в одну (деревья поиска, которые можно сливать).

4. Функциональная. Функциональные структуры данных полностью персистентны по определению, так как в них запрещаются уничтожающие присваивания, т.е. любой переменной значение может быть присвоено только один раз и изменять значения переменных нельзя. Если структура данных функциональна, то она и конфлюэнтна, если конфлюэнтна, то и полностью персистентна, если полностью персистентна, то и частично персистентна. Однако бывают структуры данных не функциональные, но конфлюэнтные.

Для решения этой задачи я использовал персистентное дерево отрезков. Само по себе дерево отрезков представляет собой сбалансировнаное бинарное дерево, хранящее на узлах диапазоны индексов. Структура дерева такова, что корень хранит в себе весь диапазон индексов из массива, а его потоки хранят полудиапазоны, разделенные по центру и так далее ниже, пока размерность диапазона не составит единицу. Также узлы такого дерева имеют счетчики, благодаря которым можно, например, легко выполнить запрос суммы на каком-то интервале массива. Именно это нам и потребуется для решения задачи.

Для реализации персистентности мы должны ввести такое понятие, как событие. С каждым введенным отрезком будет связяно 2 события, то есть всегда будем иметь $2 * n$ событий, где n - количество отрезков. Первое событие - добавление отрезка в дерева, оно имеет вид (left, type, y), где left - левая граница отрезка по координате x, type - тип события, y - координата y отрезка. Второе событие - удаление отрезка из дерева - имеет вид (right, type, y), где right - правая граница отрезка по координате x, type - тип события, y - координата y отрезка. Всего бывают два типа событий - (1 - добаление отрезка; -1 - удаление). Сначала нужно собрать все уникальные высоты

отрезков в отсортированный массив и по нему построить дерево отрезков нулевой версии.

После чего нужно отсортировать все события по первому значению (если они совпадают, то по второму значению, но уже по убыванию). Далее для каждого события строить новую версию дерева отрезков, изменяя в новой версии счетчик узлов, хранящих в себе диапазон индексов, в который входит высота текущего события. Далее после построения обрабатывать запросы для точек и обращаться к нужной версии дерева, для ответа запрашивая сумму счетчиков на нужном диапазоне.

2 Исходный код

Приложен исходный код программы

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <tuple>

struct PersistentSegmentTree {
    struct Node {
        int count;
        Node* left;
        Node* right;
        Node(int count = 0, Node* left = nullptr, Node* right = nullptr)
            : count(count), left(left), right(right) {}
    };

    Node* build(int l, int r) {
        if (l == r) {
            return new Node();
        }
        int mid = (l + r) / 2;
        return new Node(0, build(l, mid), build(mid + 1, r));
    }

    Node* update(Node* node, int l, int r, int pos, int value) {
        if (l == r) {
            return new Node(node->count + value);
        }
        int mid = (l + r) / 2;
        if (pos <= mid) {
            return new Node(node->count + value, update(node->left, l, mid, pos, value)
                , node->right);
        } else {
            return new Node(node->count + value, node->left, update(node->right, mid +
                1, r, pos, value));
```

```
        }
    }

    int query(Node* node, int l, int r, int ql, int qr) {
        if (!node || ql > r || qr < l) {
            return 0;
        }
        if (ql <= l && r <= qr) {
            return node->count;
        }
        int mid = (l + r) / 2;
        return query(node->left, l, mid, ql, qr) + query(node->right, mid + 1, r, ql,
            qr);
    }
};

int main() {
    int n, m;
    std::cin >> n >> m;

    std::vector<std::tuple<int, int, int>> events;
    std::vector<int> ys;

    for (int i = 0; i < n; ++i) {
        int l, r, h;
        std::cin >> l >> r >> h;
        events.emplace_back(l, 1, h);
        events.emplace_back(r, -1, h);
        ys.push_back(h);
    }

    std::vector<std::pair<int, int>> points(m);
    for (int i = 0; i < m; ++i) {
        std::cin >> points[i].first >> points[i].second;

    }

    sort(ys.begin(), ys.end());
    ys.erase(unique(ys.begin(), ys.end()), ys.end());
    auto getCompressedY = [&](int y) {
        return lower_bound(ys.begin(), ys.end(), y) - ys.begin();
    };

    std::sort(events.begin(), events.end(), [](const auto& a, const auto& b) {
        if (std::get<0>(a) != std::get<0>(b)) {
            return std::get<0>(a) < std::get<0>(b);
        }
        return std::get<1>(a) > std::get<1>(b);
    });
```

```
80
81      std::vector<int> queryOrder(m);
82      for (int i = 0; i < m; ++i) {
83          queryOrder[i] = i;
84      }
85      std::sort(queryOrder.begin(), queryOrder.end(), [&](int a, int b) {
86          return points[a].first < points[b].first;
87      });
88
89      PersistentSegmentTree pst;
90      std::vector<PersistentSegmentTree::Node*> roots;
91      roots.push_back(pst.build(0, ys.size() - 1));
92
93
94      for (const auto& [x, type, value] : events) {
95          if (type == 1) {
96              int h = getCompressedY(value);
97              roots.push_back(pst.update(roots.back(), 0, ys.size() - 1, h, 1));
98          } else if (type == -1) {
99              int h = getCompressedY(value);
100             roots.push_back(pst.update(roots.back(), 0, ys.size() - 1, h, -1));
101         }
102     }
103
104
105     std::vector<int> results(m);
106     int currentEvent = 0;
107     bool flag = false;
108
109     for (int i : queryOrder) {
110         int px = points[i].first;
111         int py = points[i].second;
112         int compressedY = getCompressedY(py);
113
114         while (currentEvent < events.size() && std::get<0>(events[currentEvent]) < px)
                {
115             ++currentEvent;
116         }
117
118         int targetEvent = currentEvent;
119         while (targetEvent < events.size() && std::get<0>(events[targetEvent]) == px) {
120             if (std::get<1>(events[targetEvent]) == -1) {
121                 break;
122             }
123             ++targetEvent;
124         }
125
126         int version = targetEvent;
127         if (py == ys[compressedY]) {
```

```
128             results[i] = pst.query(roots[version], 0, ys.size() - 1, compressedY + 1,
                    ys.size() - 1);
129         }
130         else {
131             results[i] = pst.query(roots[version], 0, ys.size() - 1, compressedY, ys.
                    size() - 1);
132         }
133     }
134
135     for (int res : results) {
136         std::cout << res << "\n";
137     }
138
139     return 0;
140 }
```

3 Консоль

```
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/kp$ g++ main_pers.cpp
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/kp$ ./a.out
4 2
3 8 1
0 5 2
5 11 3
2 8 4
5 -2
2 1
4
2
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/kp$ ./a.out
5 3
0 5 2
1 3 1
1 8 5
-2 0 3
-1 6 -1
1 0
3 -3
3 5
3
4
0
```

Сложность алгоритма:

Сложность построения дерева - $O(k)$, где k - количество уникальных высот (размер массива)
Сложность обновления дерева - $O(log(k))$, где k - количество уникальных высот (размер массива)
Сложность запроса в версию - $O(log(k))$, где k - количество уникальных высот (размер массива)

4 Тест производительности

В тестах посмотрим на то, как быстро алгоритм сможет обработать определенное количество запросов на определенном количестве отрезков. Для тестов будут использованы следующие пары n, m: (10000, 1000); (1000000, 10); (100, 100000); (1000000, 100000).

```
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/kp$ g++ bench_pers.cpp
.potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/kp$ ./a.out
10000 1000
10000 segments and 1000 points are processed in 32 ms
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/kp$ ./a.out
1000000 10
1000000 segments and 10 points are processed in 3347 ms
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/kp$ ./a.out
100 100000
100 segments and 100000 points are processed in 57 ms
potatogrill24@DESKTOP-7CM71EV:~/progs/Diskran/kp$ ./a.out
1000000 100000
1000000 segments and 100000 points are processed in 37075 ms
```

Выводы по производительности

Сложность обновлений дерева и запросов в него логарифмическая, потому что при обновлении мы проходим путь от корня до листа, то есть один раз проходим дерево вглубь, а глубина бинарного дерева - логарифм, то же самое и с запросом в него, когда мы получаем запрос, мы переходим от корня к детям и считываем сумму до корня, переходя к нужному потомку каждый раз. Построение дерева линейное, что логично, ведь мы имеем всегда n листьев и примерно $2n - 1$ узлов в нем, как и в бинарном сбалансированном дереве.

5 Выводы

В ходе выполнения курсового проекта, я разобрался со структурой данных, которая называется дерево отрезков, рассмотрел персистентный его вариант. Также я применил метод сканирующей прямой для решения этой задачи, который оказался весьма эффективен. Считаю, что эти знания мне пригодятся в будущем для написание интересных и сложных программ.

Список литературы

[1] *Персистентные структуры данных*
URL: https://neerc.ifmo.ru/wiki/index.php?title=Персистентные_структуры_данных

[2] *Дерево отрезков*
URL: https://algorithmica.org/ru/segtree

[3] *Сканирующая прямая*
URL: https://ru.algorithmica.org/cs/decomposition/scanline/