Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовая работа на тему

Пространственный поиск

Студент: И. С. Глушатов
Преподаватель: С. А. Сорокин
Группа: М8О-207Б-19
Дата:
Оценка:
Подпись:

Москва, 2021

Задача

Задача: Реализовать систему для определения принадлежности точки одному из многоугольников на плоскости.

**./prog index --input <input file> **
--output <index file>

Ключ	Значение
--input	входной файл с многоугольниками
--output	выходной файл с индексом

**./prog search --index <index file> **
**--input <input file> **
--output <output file>

Ключ	Значение
--index	входной файл с индексом
--input	входной файл с запросами
--output	выходной файл с ответами на запросы

Формат входного файла:

<количество многоугольников> <количество вершин многоугольника [n]>
$<x_1>$ $<y_1>$ $<x_2>$ $<y_2>$ **...** $<x_n>$ $<y_n>$

Формат файла запросов:

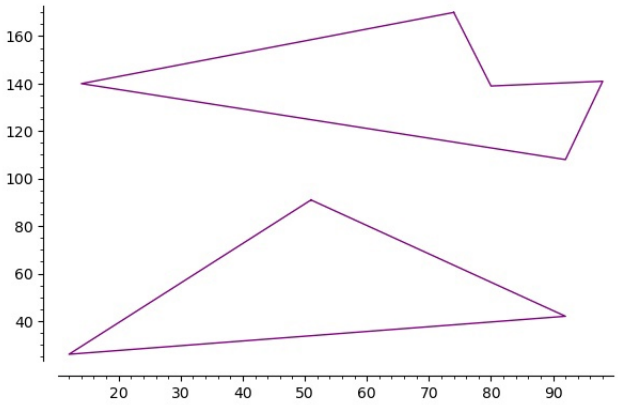
$<x_i>$ $<y_i>$

Для каждого запроса вывести номер многоугольника, внутри которого содержится точка (многоугольники нумеруются с единицы), либо -1.

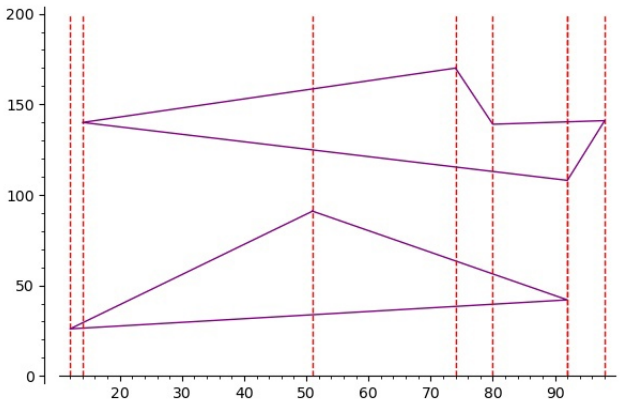
1 Описание

Задача о принадлежности точки многоугольнику является фундаментальной темой в вычислительной геометрии. Существует много алгоритмов для решени данной задачи, такие как метод трассировки лучей, учёт числа оборотов, суммирование углов. Все эти алгоритмы работают без предварительной обработки фигуры. В данной работе мне пришлось работать с препроцессингом, использующим персистентную структуру данных (сбалансированное AVL-дерево).

Допустим, что у нас есть фигуры, изображенные на рис. 1.



(a) рис. 1

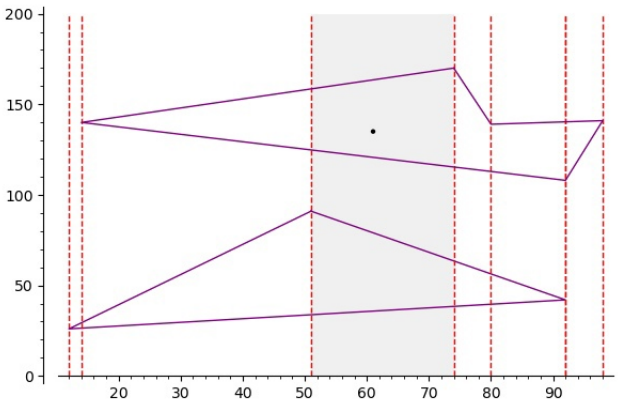


(b) рис. 2

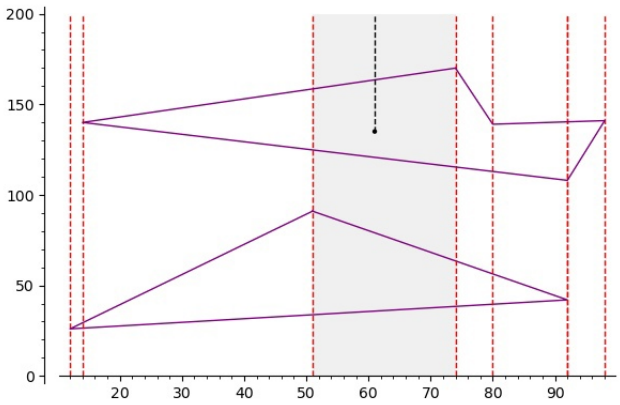
Тогда всю плоскость можно мысленно разбить на плиты (slabs в английской литературе), проведя через каждую точку вертикальную прямую (рис. 2).

Если многоугольники без самопересечений, то мы можем гарантировать, что внутри каждой плиты отрезки, полученные разбиением ребер можно упорядочить (определить оператор отношения).

Вообще если брать произвольные отрезки, то упорядочить их невозможно, однако разбиение на плиты позволяет нам это сделать. Таким образом сложность поиска должна составить $O(log_2^2(n))$



(c) рис. 3



(d) рис. 4

Подав точку на вход мы можем бинарным поиском найти нужную нам плиту (рис. 3) (для этих целей служит отсортированный массив x'ов), после чего в дереве искать место, между какими ребрами попала точка. Если в процессе препроцессинга хранить в дереве количество ребер выше (т.е. в правом поддереве), то можно за логарифмическую сложность узнать, сколько ребер находится над точкой и по методу

трассировки лучей (рис. 4), выдать ответ о принадлежности точки многоугольнику. Номер многоугольника выбирается за счет хранения и запоминания в процессе поиска в дереве номера фигуры соответствующего пройденному ребру.

Сложность препроцессинга же $O(nlog(n))$, что можно будет заметить на тестах.

2 Исходный код

Классы точки и ребра - простые библиотеки с множеством методов и конструкторов, позволяющие абстрактно использовать их в ходе всего курсового проекта. Так как в основном вся реализация простая, то я не буду включать их в листинг курсовой.

```
1
2  #pragma once
3  #include <iostream>
4  #include <math.h>
5  #include <cfloat>
6  #include <fstream>
7
8  class Point {
9    public:
10     double x;
11     double y;
12     Point(double _x = 0.0, double _y = 0.0);
13     Point operator+ (const Point&) const;
14     Point operator- (const Point&) const;
15     friend Point operator* (const double, const Point&);
16     double operator[] (int);
17     bool operator== (Point) const;
18     bool operator< (Point) const;
19     bool operator> (Point) const;
20     bool operator!= (Point) const;
21     int classify(Point, Point);
22     double polarAngle(void);
23     double length(void);
24     double distance(Point&, Point&);
25     friend std::ostream& operator<<(std::ostream&, const Point&);
26 };
27
28 enum {
29   LEFT,
30   RIGHT,
31   BEYOND,
32   BEHIND,
33   BETWEEN,
34   ORIGIN,
35   DESTINATION
36 };
37
38 class Edge {
39   public:
40     Point org;
41     Point dest;
42     Edge(void);
43     Edge(double p1_x, double p1_y, double p2_x, double p2_y);
44     Edge(Point& _org, Point& _dest);
45     Edge(Point& _org, Point&& _dest);
46     Edge& rot(void);
47     Edge& flip(void);
48     Point point(double);
49     int intersect(Edge&, double&);
50     int intersect(Edge&&, double&);
51     int cross(Edge&, double&);
52     bool isVertical(void);
53     double slope(void) const;
54     double y(double) const;
55     bool abovePoint(Point) const;
```

```
    bool underPoint(Point) const;
    bool crossPoint(Point) const;
    bool operator==(Edge) const;
    bool operator!=(Edge) const;
    double min_y() const;
    double max_y() const;
    friend std::ofstream& operator<<(std::ofstream&, const Edge&);
    friend std::ostream& operator<<(std::ostream&, const Edge&);
    friend std::ifstream& operator>>(std::ifstream&, Edge&);
};

enum {
  COLLINEAR,
  PARALLEL,
  SKEW,
  SKEW_CROSS,
  SKEW_NO_CROSS
};
```

Самый важный класс из курсового проекта - персистентное сбалансированное AVL-дерево. О методах подробнее в таблице:

persistent_tree2.hpp	
PersistentTree::Insert (const K&, const V&, const double, const bool)	Вставка в персистентное дерево. Булевый флаг указывает, сохранять ли новую версию дерева или нет
PersistentTree::Remove (const K&, const double, const bool)	Удаление из дерева по ключу с такой же функцией у булевого флага
PersistentTree::NotChange ()	Копирование самой последней версии
PersistentTree::FindNumberAbove (const unsigned int, const Point&, long*, long*)	Возвращает количество рёбер над точкой
PersistentTree::Print (...)	Вывод дерева в консоль
PersistentTree::operator« (ofstream&, PersistentTree&)	Сохранение дерева в файл
PersistentTree::operator» (ifstream&, PersistentTree&)	Загрузка дерева из файла

```
#pragma once

#include <iostream>
#include <vector>
#include <memory>
#include <map>
#include <unordered_map>
#include <queue>
#include "utilitys.hpp"

using namespace std;

static unsigned long long index = 1;

template<class K, class V>
struct PersistentTree {

  struct node {

    using h_type = unsigned int;
    using nre_type = unsigned int;
```

```
22      using idx_type = unsigned long long;
23      using node_ptr = shared_ptr<node>;
24
25      node_ptr l = nullptr;
26      node_ptr r = nullptr;
27      K key;
28      V value;
29      h_type h;
30      nre_type nre;
31      idx_type idx;
32
33      node(const K& _k, const V& _v): key(_k), value(_v), h(1), nre(0) {
34        idx=index;
35        index++;
36      }
37      node(const K& _k, const V& _v, const h_type& _h, const nre_type& _nre): key(_k),
            value(_v), h(_h), nre(_nre) {
38        idx=index;
39        index++;
40      }
41
42      node_ptr RightRotate(node_ptr head) {
43        node_ptr temp1 = make_shared<node>(head->l->key, head->l->value, head->l->h, 1 +
              head->l->nre + head->nre);
44        node_ptr temp2 = make_shared<node>(head->key, head->value, head->h, head->nre);
45        temp1->l = head->l->l;
46        temp1->r = temp2;
47        temp2->l = head->l->r;
48        temp2->r = head->r;
49        FixHeight(temp2);
50        FixHeight(temp1);
51        return temp1;
52      }
53
54      node_ptr LeftRotate(node_ptr head) {
55
56        if (head->key == head->r->key)
57          return head;
58
59        node_ptr temp1 = make_shared<node>(head->r->key, head->r->value, head->r->h, head
              ->r->nre);
60        node_ptr temp2 = make_shared<node>(head->key, head->value, head->h, head->nre - 1
               - head->r->nre);
61        temp1->l = temp2;
62        temp1->r = head->r->r;
63        temp2->l = head->l;
64        temp2->r = head->r->l;
65        FixHeight(temp2);
66        FixHeight(temp1);
67        return temp1;
68      }
69
70      node_ptr Balancing(node_ptr head) {
71        node_ptr temp = make_shared<node>(head->key, head->value, head->h, head->nre);
72        temp->l = head->l;
73        temp->r = head->r;
74        FixHeight(temp);
75        if (Balance(temp)==2) {
76          if (Balance(temp->l)<0) {
77            temp->l = LeftRotate(head->l);
78          }
```

```
79          return RightRotate(temp);
80        } else if (Balance(temp)==-2) {
81          if (Balance(temp->r)>0) {
82            temp->r = RightRotate(head->r);
83          }
84          return LeftRotate(temp);
85        }
86        return temp;
87      }
88
89      node_ptr Insert(const node_ptr parent, const K& key, const V& value, const double
            slab) {
90        static unsigned int index = 1;
91        if (parent) {
92          node_ptr temp;
93          if (key.y(slab) > parent->key.y(slab)) {
94            temp = make_shared<node>(parent->key, parent->value, parent->h, parent->nre +
                   1);
95            temp->l = parent->l;
96            temp->r = Insert(parent->r, key, value, slab);
97          } else {
98            temp = make_shared<node>(parent->key, parent->value, parent->h, parent->nre);
99            temp->r = parent->r;
100           temp->l = Insert(parent->l, key, value, slab);
101         }
102
103         return Balancing(temp);
104       }
105
106       return make_shared<node>(key, value);
107     }
108
109     node_ptr MinRight(node_ptr head) {
110       node_ptr temp = head;
111       while (temp->l) {
112         temp = temp->l;
113       }
114       return temp;
115     }
116
117     node_ptr RemoveMin(node_ptr head) {
118       if (!head->l) {
119         node_ptr temp = head->r;
120         return temp;
121       }
122       node_ptr temp = make_shared<node>(head->key, head->value, head->h, head->nre);
123       temp->r = head->r;
124       temp->l = RemoveMin(head->l);
125       return Balancing(temp);
126     }
127
128     node_ptr Remove(node_ptr parent, const K& key, const double slab) {
129       if (!parent)
130         return nullptr;
131
132       node_ptr temp;
133       if (key == parent->key) {
134         if (!parent->l and !parent->r) {
135           return nullptr;
136         }
137
```

```
138         if (!parent->r) {
139           return parent->l;
140         }
141
142         node_ptr min_right = MinRight(parent->r);
143         temp = make_shared<node>(min_right->key, min_right->value, parent->h, parent->
                nre - 1);
144         temp->l = parent->l;
145         temp->r = RemoveMin(parent->r);
146       } else if (key.y(slab) < parent->key.y(slab)) {
147         temp = make_shared<node>(parent->key, parent->value, parent->h, parent->nre);
148         temp->r = parent->r;
149         temp->l = Remove(parent->l, key, slab);
150       } else {
151         temp = make_shared<node>(parent->key, parent->value, parent->h, parent->nre -
                1);
152         temp->l = parent->l;
153         temp->r = Remove(parent->r, key, slab);
154       }
155
156       return Balancing(temp);
157     }
158
159     nre_type FindNumberAbove(const node_ptr head, const Point& p, bool *onEdge, long*
            left_ancestor, long* right_ancestor) {
160       if (head) {
161         if (p.y > head->key.y(p.x)) {
162           *left_ancestor = static_cast<long>(head->value);
163           return FindNumberAbove(head->r, p, onEdge, left_ancestor, right_ancestor);
164         } else if (p.y == head->key.y(p.x)) {
165           *left_ancestor = static_cast<long>(head->value);
166           *right_ancestor = static_cast<long>(head->value);
167           *onEdge = true;
168           return 0;
169         } else {
170           *right_ancestor = static_cast<long>(head->value);
171           return head->nre + 1 + FindNumberAbove(head->l, p, onEdge, left_ancestor,
                    right_ancestor);
172         }
173       }
174
175       return 0;
176     }
177
178     h_type Height(const node_ptr head) const {
179       return head ? head->h : 0;
180     }
181
182     int Balance(const node_ptr head) const {
183       return head ? Height(head->l)-Height(head->r) : 0;
184     }
185
186     void FixHeight(node_ptr head) {
187       head->h = (Height(head->l)>Height(head->r) ? Height(head->l) : Height(head->r))
              +1;
188     }
189
190     void Print(const node_ptr head, unsigned int tab) const {
191       if (head) {
192         Print(head->r, tab + 1);
193         for (unsigned int i = 0; i < tab; i++) std::cout << "\t";
```

```
194         std::cout << "(" << head->key << ", " << head->value << ", " << head->h << ", "
                << head->nre << ", " << head->idx << ")\n";
195         Print(head->l, tab + 1);
196       }
197     }
198
199     friend std::ostream& operator<<(std::ostream& out, const node& p) {
200       out << p.key << " " << p.value << " " << p.h << " " << p.nre << " " << p.idx;
201       return out;
202     }
203
204     friend std::ofstream& operator<<(std::ofstream& out, const node& p) {
205       out << p.key << " " << p.value << " " << p.h << " " << p.nre << " " << p.idx;
206       return out;
207     }
208
209   };
210
211   using node_ptr = shared_ptr<node>;
212
213   unsigned int number_of_versions = 0;
214   vector<node_ptr> trees;
215
216   void Insert(const K& key, const V& value, const double slab, const bool flag = false
        ) {
217     if (!flag) {
218       if (trees.empty()) {
219         trees.push_back(nullptr);
220         trees[0] = make_shared<node>(key, value);
221       } else {
222         trees.push_back(nullptr);
223         trees[number_of_versions] = trees[number_of_versions]->Insert(trees[
              number_of_versions - 1], key, value, slab);
224       }
225       number_of_versions++;
226     } else {
227       if (trees.empty()) {
228         trees.push_back(nullptr);
229         trees[0] = make_shared<node>(key, value);
230         number_of_versions++;
231       } else {
232         trees[number_of_versions - 1] = trees[number_of_versions - 1]->Insert(trees[
              number_of_versions - 1], key, value, slab);
233       }
234     }
235   }
236
237   void Remove(const K& key, const double slab, const bool flag = false) {
238     if (!flag) {
239       trees.push_back(nullptr);
240       trees[number_of_versions] = trees[number_of_versions]->Remove(trees[
            number_of_versions - 1], key, slab);
241       number_of_versions++;
242     } else {
243       trees[number_of_versions - 1] = trees[number_of_versions - 1]->Remove(trees[
            number_of_versions - 1], key, slab);
244     }
245   }
246
247   void NotChange() {
248     trees.push_back(nullptr);
```

```
249     trees[number_of_versions] = trees[number_of_versions - 1];
250     number_of_versions++;
251   }
252
253   unsigned int FindNumberAbove(const unsigned int version, const Point& p, long*
          left_ancestor, long* right_ancestor) {
254     if (version >= trees.size()) {
255       print("bad version: ", version);
256       throw "There is no such version";
257     }
258
259     bool onEdge = false;
260     unsigned int result = trees[version]->FindNumberAbove(trees[version], p, &onEdge,
            left_ancestor, right_ancestor);
261
262     if (onEdge)
263       result = 1;
264
265     return result;
266   }
267
268   void Print(const unsigned int version) const {
269     if (version >= trees.size()) {
270       cout << "|--------------------------------------------------|" << endl;
271       cout << "There is no such version of tree" << endl;
272       cout << "|--------------------------------------------------|" << endl;
273       return;
274     }
275
276     cout << "|--------------------------------------------------|" << endl;
277     cout << "[" << version << "]" << endl;
278     trees[version]->Print(trees[version], 0);
279     cout << "|--------------------------------------------------|" << endl;
280   }
281
282   void Print() const {
283     unsigned int v = 1;
284     cout << number_of_versions << " versions in total:\n";
285     for (node_ptr tree : trees) {
286       cout << "|--------------------------------------------------|" << endl;
287       cout << "[" << v << "]" << endl;
288       tree->Print(tree, 0);
289       cout << "|--------------------------------------------------|" << endl;
290       v++;
291     }
292   }
293
294   friend ofstream& operator<<(ofstream& out, PersistentTree& pt) {
295     map<unsigned long long, node> visited;
296
297     node_ptr curr;
298     out << pt.trees.size() << "\n";
299
300     for (size_t i = 0; i < pt.trees.size(); i++) {
301       curr = pt.trees[i];
302
303       if (curr == nullptr) {
304         out << "0 0 0 0 0 0 0 0 ";
305         continue;
306       } else {
307         out << *curr << " ";
```

```
    }

    queue<node_ptr> visited_in_this_tree;

    if (curr->l != nullptr)
      visited_in_this_tree.push(curr->l);

    if (curr->r != nullptr)
      visited_in_this_tree.push(curr->r);

    while (!visited_in_this_tree.empty()) {
      curr = visited_in_this_tree.front();
      visited_in_this_tree.pop();

      visited.insert({curr->idx, *curr});

      if (curr->l != nullptr)
        visited_in_this_tree.push(curr->l);

      if (curr->r != nullptr)
        visited_in_this_tree.push(curr->r);
    }
  }

  out << "\n" << visited.size() << "\n";

  for (const auto n : visited) {
    out << n.second << " ";
  }

  out << "\n";

  for (size_t i = 0; i < pt.trees.size(); i++) {
    curr = pt.trees[i];

    if (curr == nullptr)
      continue;

    queue<node_ptr> visited_in_this_tree;
    visited_in_this_tree.push(curr);

    while (!visited_in_this_tree.empty()) {
      curr = visited_in_this_tree.front();
      visited_in_this_tree.pop();

      if (curr->l != nullptr) {
        out << curr->idx << " " << curr->l->idx << " " << "0" << " ";
        visited_in_this_tree.push(curr->l);
      }

      if (curr->r != nullptr) {
        out << curr->idx << " " << curr->r->idx << " " << "1" << " ";
        visited_in_this_tree.push(curr->r);
      }
    }
  }

  return out;
}
```

```
369  friend ifstream& operator>>(ifstream& in, PersistentTree& pt) {
370
371    in >> pt.number_of_versions;
372    for (size_t i = 0; i < pt.number_of_versions; i++) {
373      Edge key;
374      unsigned int value;
375      unsigned int h;
376      unsigned int nre;
377      unsigned long long idx;
378
379      in >> key >> value >> h >> nre >> idx;
380      node_ptr np = make_shared<PersistentTree<K, V>::node>(key, value, h, nre);
381      np->idx = idx;
382      if (idx == 0) np = nullptr;
383      pt.trees.push_back(np);
384    }
385
386    unsigned int number_of_inside_vertexes;
387    in >> number_of_inside_vertexes;
388
389    vector<node_ptr> inside_vertexes;
390
391    for (size_t i = 0; i < number_of_inside_vertexes; i++) {
392      Edge key;
393      unsigned int value;
394      unsigned int h;
395      unsigned int nre;
396      unsigned long long idx;
397
398      in >> key >> value >> h >> nre >> idx;
399      node_ptr np = make_shared<PersistentTree<K, V>::node>(key, value, h, nre);
400      np->idx = idx;
401      inside_vertexes.push_back(np);
402    }
403
404    unsigned long long num_from, num_to, side;
405    while (in >> num_from >> num_to >> side) {
406      node_ptr from = nullptr;
407      node_ptr to = nullptr;
408
409      for (size_t i = 0; i < inside_vertexes.size(); i++) {
410        if (inside_vertexes[i]->idx == num_from) {
411          from = inside_vertexes[i];
412        }
413
414        if (inside_vertexes[i]->idx == num_to) {
415          to = inside_vertexes[i];
416        }
417      }
418
419      if (from == nullptr) {
420        for (size_t i = 0; i < pt.trees.size(); i++) {
421          if (pt.trees[i] == nullptr)
422            continue;
423
424          if (pt.trees[i]->idx == num_from) {
425            from = pt.trees[i];
426          }
427        }
428      }
429
```

```
      if (side == 0) {
        from->l = to;
      } else if (side == 1) {
        from->r = to;
      }

    }

    return in;
  }

};
```

Основной prog2.cpp файл с программой и классом Index.

```
#include <iostream>
#include <cstdlib>
#include <climits>
#include <ctime>
#include <vector>
#include <algorithm>
#include <fstream>
#include <chrono>
#include "point.hpp"
#include "edge.hpp"
#include "persistent_tree2.hpp"

// #define PRINT
// #define TREE
// #define LOG

using namespace std;

struct Index {

  PersistentTree<Edge, unsigned int> tree;
  vector<double> slabs;

  Index() = default;

  Index(vector<vector<Point>> figures) {

    vector<pair<Edge, unsigned int>> edges;
    vector<Edge> for_enter;

#ifdef PRINT
    print("[");
    for (int i = 0; i < figures.size(); i++){
      print(" ", figures[i]);
    }
    print("]");
#endif

    for (vector points : figures) {
      static unsigned int figure_number = 1;
      for (size_t i = 0; i < points.size(); i++) {
        if (points[i].x == points[(i+1) % points.size()].x) continue;
        edges.push_back(pair<Edge, unsigned int> (Edge(points[i], points[(i+1) % points
            .size()]), figure_number) );
        for_enter.push_back(Edge(points[i], points[(i+1) % points.size()]));
      }
      figure_number++;
```

```
47      }
48
49
50      for (vector points : figures) {
51        for (const Point p : points) {
52          if (find(slabs.begin(), slabs.end(), p.x) == slabs.end())
53            slabs.push_back(p.x);
54        }
55      }
56
57
58      sort(slabs.begin(), slabs.end());
59      sort(edges.begin(), edges.end(), [](const pair<Edge, unsigned int>& p1, const pair<
            Edge, unsigned int>& p2) {
60        return p1.first.org.x < p2.first.org.x;
61      });
62      sort(for_enter.begin(), for_enter.end(), [](const Edge& p1, const Edge& p2) {
63        return p1.dest.x < p2.dest.x;
64      });
65
66      bool flag = false;
67      for (int i = 0; i < slabs.size(); i++) {
68
69
70        while (!for_enter.empty() and for_enter[0].dest.x == slabs[i] and i>0) {
71          tree.Remove(for_enter[0], slabs[i-1] + 10E-9, flag);
72          for_enter.erase(for_enter.begin());
73
74          if (!flag)
75            flag = true;
76        }
77
78
79        while (!edges.empty() and edges[0].first.org.x == slabs[i]) {
80          tree.Insert(edges[0].first, edges[0].second, slabs[i] + 10E-9, flag);
81          edges.erase(edges.begin());
82
83          if (!flag)
84            flag = true;
85        }
86
87
88        if (!flag)
89          tree.NotChange();
90
91        flag = false;
92      }
93  #ifdef PRINT
94  #ifdef TREE
95      tree.Print();
96  #endif
97  #endif
98    }
99
100   unsigned int NumberOfEdgesAbovePoint(Point p, long* la, long* ra) {
101
102     if (p.x < slabs[0] or p.x > slabs[slabs.size()-1])
103       return 0;
104     unsigned int l = 0, r = slabs.size();
105
106     bool flag = false;
```

```
107     while(r - l > 1) {
108
109       unsigned int mid = (l + r) / 2;
110
111       if (slabs[mid] == p.x) {
112         l = mid;
113         flag = true;
114         break;
115       }
116
117       if (p.x < slabs[mid]) {
118         r = mid;
119       } else {
120         l = mid;
121       }
122
123     }
124
125     long left_ancestor = -1, right_ancestor = -1;
126     unsigned int res = tree.FindNumberAbove(l > 0 ? l-flag : l, p, &left_ancestor, &
          right_ancestor);
127
128     *la = left_ancestor;
129     *ra = right_ancestor;
130     return res;
131   }
132
133   long inWhichPoligone(Point p) {
134
135     long left_ancestor = -1, right_ancestor = -1;
136     unsigned int noeap = NumberOfEdgesAbovePoint(p, &left_ancestor, &right_ancestor);
137
138     if (noeap % 2 == 1) {
139       return left_ancestor;
140     }
141
142     return -1;
143   }
144
145   void Write(string name_of_file) {
146     ofstream save_file;
147     save_file.open(name_of_file);
148
149     if (!save_file.is_open()) {
150       print("Bad record");
151     }
152
153     save_file << slabs.size() << "\n";
154     for (const double& p : slabs) {
155       save_file << p << " ";
156     }
157     save_file << "\n";
158     save_file << tree << "\n";
159
160     save_file.close();
161   }
162
163   void Read(string name_of_file) {
164     ifstream save_file;
165     save_file.open(name_of_file);
166
```

```
    if (!save_file.is_open()) {
      print("Bad reading");
    }

    unsigned int size_of_slabs;
    save_file >> size_of_slabs;

    for (size_t i = 0; i < size_of_slabs; i++) {
      double slab;
      save_file >> slab;
      slabs.push_back(slab);
    }

    save_file >> tree;
    save_file.close();
  }

};


void Read(vector<vector<Point>>* figures, istream& from) {
  unsigned int number_of_figures;
  from >> number_of_figures;

  for (const unsigned int i : range<unsigned int>(0, number_of_figures)) {
    unsigned int number_of_vertexes;
    from >> number_of_vertexes;

    figures->push_back(vector<Point>());

    for(const unsigned int j : range<unsigned int>(0, number_of_vertexes)) {
      double l, r;
      from >> l >> r;
      (*figures)[i].push_back(Point(l, r));
    }
  }
}


int main(int args, char** argv) {
  vector<vector<Point>> figures;

#ifdef LOG
  std::chrono::time_point<std::chrono::system_clock> start;
  std::chrono::time_point<std::chrono::system_clock> end;
  ofstream log_file;
  log_file.open("log1.txt", std::fstream::app);
#endif

  if (args == 1) {
    print("No keys: index, search");
  }

  else if (args == 2 and string(argv[1]) == "index") {

    Read(&figures, cin);

    Index Poligones(figures);
    Poligones.Write("base_output.txt");
  }

```

```
228   else if (args == 4 and string(argv[1]) == "index" and string(argv[2]) == "--input")
          {
229
230     ifstream file;
231     file.open(argv[3]);
232
233     if (!file.is_open()) {
234       print("Bad reading");
235       return 0;
236     }
237
238     Read(&figures, file);
239
240 #ifdef LOG
241     start = std::chrono::system_clock::now();
242     Index Poligones(figures);
243     end = std::chrono::system_clock::now();
244     Poligones.Write("base_output.txt");
245     log_file << figures[0].size() << " " << std::chrono::duration<double>(end-start).
            count() << "\n";
246 #else
247     Index Poligones(figures);
248     Poligones.Write("base_output.txt");
249 #endif
250
251     file.close();
252   }
253
254   else if (args == 4 and string(argv[1]) == "index" and string(argv[2]) == "--output")
          {
255
256     Read(&figures, cin);
257
258     Index Poligones(figures);
259     Poligones.Write(argv[3]);
260   }
261
262   else if (args == 6 and string(argv[1]) == "index" and string(argv[2]) == "--input"
          and string(argv[4]) == "--output") {
263
264     ifstream file;
265     file.open(argv[3]);
266
267     if (!file.is_open()) {
268       print("Bad reading");
269       return 0;
270     }
271
272     Read(&figures, file);
273
274 #ifdef LOG
275     start = std::chrono::system_clock::now();
276     Index Poligones(figures);
277     end = std::chrono::system_clock::now();
278     Poligones.Write(argv[5]);
279     log_file << figures[0].size() << " " << std::chrono::duration<double>(end-start).
            count() << "\n";
280 #else
281     Index Poligones(figures);
282     Poligones.Write(argv[5]);
283 #endif
```

```
284
285     file.close();
286   }
287
288   else if (args == 2 and string(argv[1]) == "search") {
289     Read(&figures, cin);
290     Index Poligones(figures);
291
292     double x,y;
293     while (cin >> x >> y) {
294       cout << Poligones.inWhichPoligone(Point(x, y)) << endl;
295     }
296   }
297
298   else if (args == 4 and string(argv[1]) == "search" and string(argv[2]) == "--input")
          {
299     Read(&figures, cin);
300     Index Poligones(figures);
301
302     ifstream file;
303     file.open(argv[3]);
304
305     if (!file.is_open()) {
306       print("Bad reading");
307       return 0;
308     }
309
310     double x,y;
311     while (file >> x >> y) {
312       cout << Poligones.inWhichPoligone(Point(x, y)) << endl;
313     }
314
315     file.close();
316   }
317
318   else if (args == 4 and string(argv[1]) == "search" and string(argv[2]) == "--index")
          {
319     Index Poligones;
320     Poligones.Read(string(argv[3]));
321
322     double x,y;
323     while (cin >> x >> y) {
324       cout << Poligones.inWhichPoligone(Point(x, y)) << endl;
325     }
326
327   }
328
329   else if (args == 4 and string(argv[1]) == "search" and string(argv[2]) == "--output"
        ) {
330     Read(&figures, cin);
331     Index Poligones(figures);
332
333     ofstream file;
334     file.open(argv[3]);
335
336     if (!file.is_open()) {
337       print("Bad opening");
338       return 0;
339     }
340
341     double x,y;
```

```
342     while (cin >> x >> y) {
343       file << Poligones.inWhichPoligone(Point(x, y)) << endl;
344     }
345 
346     file.close();
347   }
348 
349   else if (args == 6 and string(argv[1]) == "search" and string(argv[2]) == "--index"
        and string(argv[4]) == "--input") {
350     Index Poligones;
351     Poligones.Read(string(argv[3]));
352 
353     ifstream file;
354     file.open(argv[5]);
355 
356     if (!file.is_open()) {
357       print("Bad opening");
358       return 0;
359     }
360 
361     double x,y;
362     while (file >> x >> y) {
363       cout << Poligones.inWhichPoligone(Point(x, y)) << endl;
364     }
365 
366     file.close();
367   }
368 
369   else if (args == 8 and string(argv[1]) == "search" and string(argv[2]) == "--index"
        and string(argv[4]) == "--input" and string(argv[6]) == "--output") {
370     Index Poligones;
371     Poligones.Read(string(argv[3]));
372 
373     ifstream file;
374     ofstream out_file;
375     file.open(argv[5]);
376     out_file.open(argv[7]);
377 
378     if (!file.is_open() or !out_file.is_open()) {
379       print("Bad opening");
380       return 0;
381     }
382 
383 #ifdef LOG
384     start = std::chrono::system_clock::now();
385     double x,y;
386     unsigned int np = 0;
387     while (file >> x >> y) {
388       np++;
389       out_file << Poligones.inWhichPoligone(Point(x, y)) << endl;
390     }
391     end = std::chrono::system_clock::now();
392     log_file << np << " " << std::chrono::duration<double>(end-start).count() << "\n";
393 #else
394     double x,y;
395     while (file >> x >> y) {
396       out_file << Poligones.inWhichPoligone(Point(x, y)) << endl;
397     }
398 #endif
399 
400     out_file.close();
```

```
401     file.close();
402   } else {
403     print("Wrong enter");
404   }
405
406 #ifdef LOG
407   log_file.close();
408 #endif
409
410   return 0;
411 }
```

3 Консоль

```
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/kp/kp5$ ./prog2.out index
2
3
1 0
5 0
4 2
4
4 3
5 4
4 5
3 4
Структура построена и сохранена в файле: base_output.txt
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/kp/kp5$ ./prog2.out search
--index base_output.txt
4 4
2
3 1
1
5 2
-1
4 2
1
1 0
1
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/kp/kp5$
```

4 Тест производительности

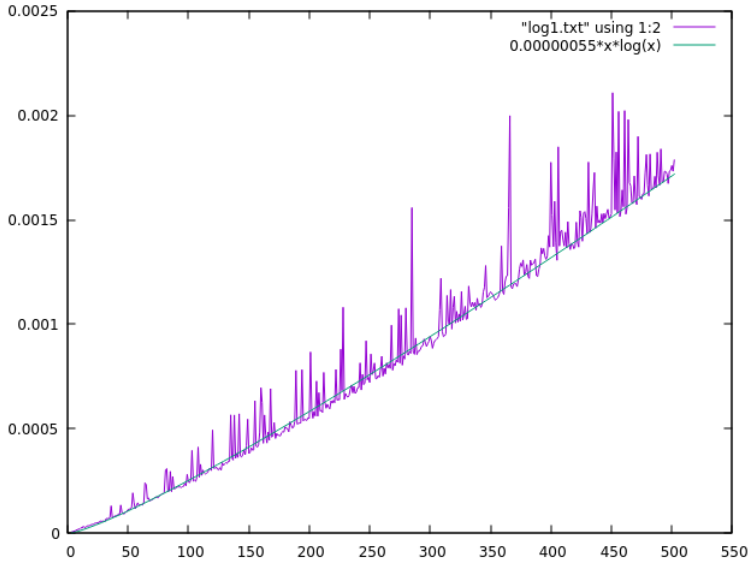


Рис. 1: График времени построения дерева от количества вершин многоугольника Ассимптотика алгоритма $O(nlog(n))$

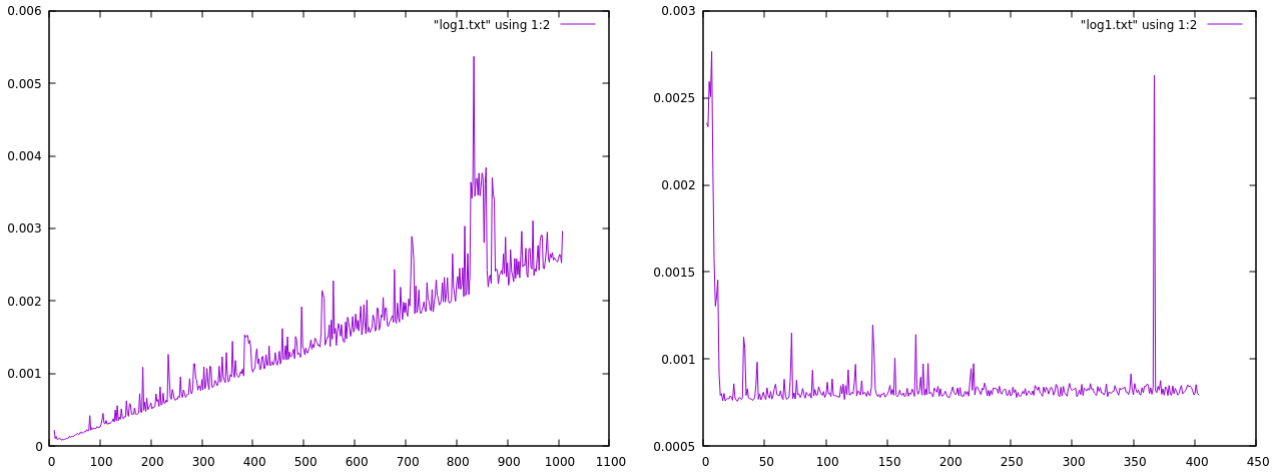


Рис. 2: Графики времени определения местоположения точки от количества точек (первый график) и от количества рёбер (второй график)

5 Выводы

В ходе курсового проекта я познакомился с персистентными структурами данных (а именно деревом), узнал о том, насколько сложно, оказывается, иметь дело с многоугольниками в вычислительной геометрии, что обосновало причину использования прямоугольной структуры приложений, в том числе и сайтов. Ознакомился с многочисленной зарубежной литературой.

Список литературы

[1] *Поисковик - Google.*
URL: `https://www.google.com/`

[2] *Сайт с подробной документацией библиотек C++*
URL: `https://en.cppreference.com/`

[3] *Persistent Data Structures*
URL: `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-854`

[4] *PLP*
URL: `https://sites.cs.ucsb.edu/ suri/cs235/Location.pdf`

[5] *Смежные задачи*
URL: `https://e-maxx.ru/algo/intersecting_segments`