МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

**"САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО"**

Институт компьютерных наук и технологий

Направление **02.03.01** : Математика и компьютерные науки

# ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

## «Диаграмма Вороного. Алгоритм построения диаграммы Вороного»

Исполнитель:____________ Яшнова Дарья Михайловна
группа 5130201/20002

Руководитель:____________ Курочкин Михаил Александрович

«____»_____________2025г

Санкт-Петербург, 2025

# Содержание

## Введение

Диаграмма Вороного конечного множества точек S на плоскости представляет такое разбиение плоскости, при котором каждая область этого разбиения образует множество точек, более близких к одному из элементов множества S, чем к любому другому элементу множества.

В данной работе необходимо изучить алгоритм построения диаграммы Вороного и реализовать его. Реализованную версию сравнить с версией из графической библиотеки.

# 1 Математическая модель

## 1.1 Определение

Пусть $S$ - множество $N$ точек на плоскости. Для каждой точки $p_i \in S$ определим область Вороного $V(i)$ как множество точек $(x, y)$ на плоскости, которые ближе к $p_i$, чем к любой другой точке в $S$. Математически:

$$V(i) = \{(x, y) \mid d((x, y), p_i) \leq d((x, y), p_j) \text{ для всех } j \neq i\}$$

где $d$ - евклидово расстояние.

Область Вороного $V(i)$ является пересечением $N - 1$ полуплоскостей:

$$V(i) = \bigcap_{j \neq i} H(p_i, p_j)$$

где $H(p_i, p_j)$ - полуплоскость, содержащая $p_i$, которая определяется перпендикулярным биссектором отрезка $\overline{p_i p_j}$.

## 1.2 Свойства

1. Области Вороного $V(p_i)$ являются выпуклыми.
2. Объединение всех областей покрывает всё пространство $R^2$.
3. Каждая вершина диаграммы Вороного является точкой пересечения в точности трех ребер диаграммы.
4. Диаграмма вороного множества из N точек имеет не более 2N-5 вершин и 3N-6 ребер.
5. Граф, двойственный диаграмме Вороного, является триангуляцией множества вершин.

## 1.3 Пример

Пример диаграммы Вороного для 10 точек представлен на рис.1:

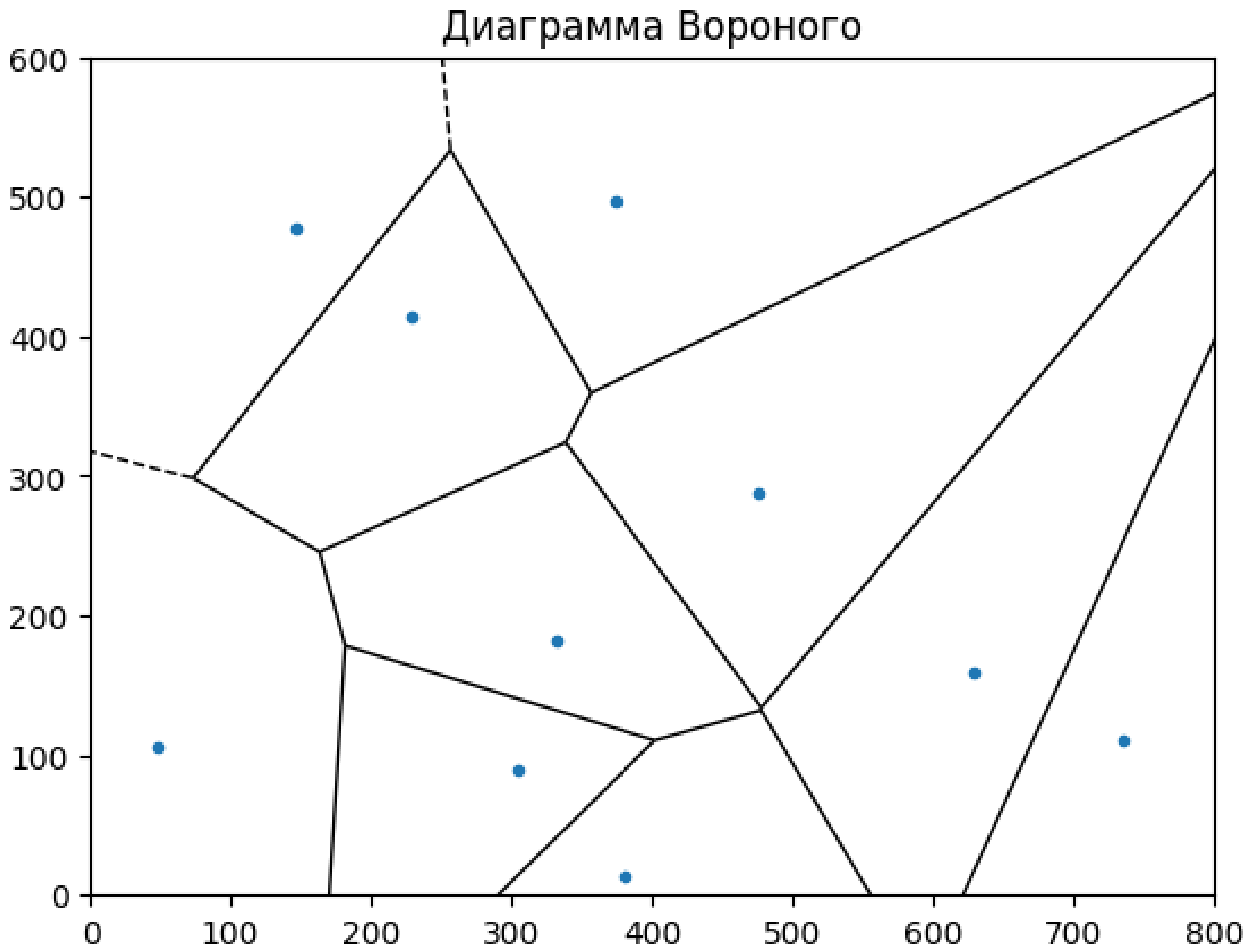



Рис. 1: Пример диаграммы Вороного для 10 точек

# 2 Алгоритмы построения диаграммы Вороного

Существует множество алгоритмов построения диаграммы Вороного, которые различаются своими идеями, скоростью работы по времени и требовательностью к количеству используемой памяти. Вот некоторые из них:

## 2.1 Алгоритм построения диаграммы Вороного «в лоб»

Данный алгоритм строит последовательно локусы для каждой p$_i$.

1. Получаем n-1 прямую (серединные перпендикуляры), так как мы провели серединные перпендикуляры всех отрезков, соединяющих данную точку p с остальными;

2.Пересекаем попарно все прямые, получаем $O(n^2)$ точек пересечения (потому что каждая прямая может пересечь все другие в худшем случае).

3.Проверяем все эти $O(n^2)$ точек на принадлежность каждой из n-1 полуплоскостей, то есть получаем уже асимптотику $O(n^3)$. Соответственно те точки, которые принадлежат всем полуплоскостям, и будут вершинами ячейки Вороного точки p;

4.Проделываем первые три шага для всех n точек, получаем итоговую асимптотику $O(n^4)$.

**Преимущества:** Простота реализации.

**Недостатки:** Очень неэффективен для больших наборов данных.

## 2.2 Алгоритм «Разделяй и властвуй»

Алгоритм «Разделяй и властвуй» является одним из наиболее эффективных алгоритмов для построения ДВ. Он состоит из следующих шагов:

1. **Разделение:** Разделить набор сайтов $S$ на два подмножества $S_1$ и $S_2$ примерно равного размера.

2. **Рекурсия:** Рекурсивно построить ДВ для $S_1$ и $S_2$.

3. **Слияние:** Объединить $\mathrm{Vor}(S_1)$ и $\mathrm{Vor}(S_2)$ для получения $\mathrm{Vor}(S)$. Этот шаг включает в себя построение разделяющей цепи $\sigma$ и удаление ненужных ребер.

**Временная сложность:** $O(N \log N)$. Шаг слияния занимает $O(N)$ времени, поэтому общая сложность определяется рекуррентным соотношением $T(N) = 2T(N/2) + O(N)$, которое решается как $O(N \log N)$.

**Преимущества:** Эффективен для больших наборов данных.

**Недостатки:** Более сложная реализация, чем наивный алгоритм.

## 2.3 Алгоритм Форчуна (Fortune's Algorithm)

Алгоритм Форчуна (также известный как алгоритм sweep line) является еще одним эффективным алгоритмом для построения диаграммы Вороного. Он использует концепцию «пляжной линии», которая представляет собой границу между областями, ближайшими к сайтам, уже пройденным sweep line, и областями, где еще не определена близость.

**Временная сложность:** $O(N \log N)$.

**Преимущества:** Эффективен и относительно прост в реализации.

**Недостатки:** Требует понимания концепции пляжной линии и обработки различных событий (site events и circle events).

## 3 Сравнение алгоритмов

| Алгоритм | Временная сложность |
|---|---|
| «В лоб» | $O(N^4)$ |
| Разделяй и властвуй | $O(N \log N)$ |
| Форчуна | $O(N \log N)$ |

## 4 Алгоритм Форчуна

В данной реализации используется алгоритм Форчуна. Алгоритм Форчуна — это метод построения диаграммы Вороного для множества точек на плоскости. Он использует сканирующую прямую, которая движется сверху вниз, и динамически обновляет структуру береговой линии, чтобы вычислить границы клеток Вороного.

**Входные данные**: Набор $P := \{p_1, p_2, \ldots, p_n\}$ точечных сайтов на плоскости.
**Выходные данные**: Диаграмма Вороного $Vor(P)$, представленная внутри рамки в виде двусвязного списка рёбер.

В данном алгоритме используется следующее определение параболы.

Парабола определяется как множество точек на плоскости, для которых расстояние до фокуса равно расстоянию до директрисы. Пусть:

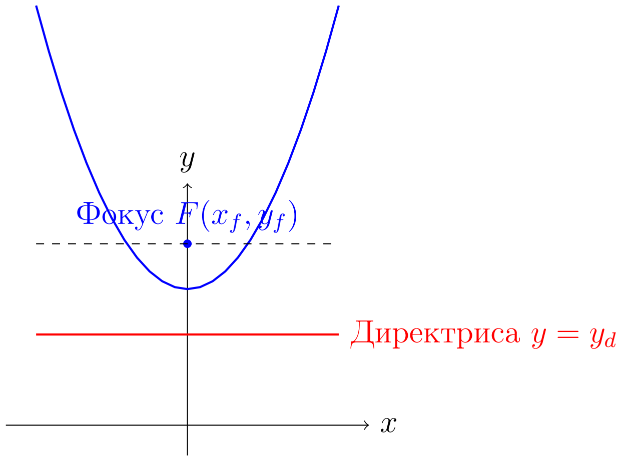
- Фокус $F(x_f, y_f)$,
- Директриса $y = y_d$, где $y_d < y_f$.

Тогда можно вывести уравнение параболы. Нам нужна формула вида $f(x) = ax^2 + bx + c$. Из определения кривой известно, что расстояние $((x, y_d), (x, f(x)))$ равно расстоянию $((x_f, y_f), (x, f(x)))$, поэтому можно сделать следующее:

$$\begin{aligned}
\sqrt{(x-x)^2 + (f(x)-y_d)^2} &= \sqrt{(x-x_f)^2 + (f(x)-y_f)^2} \\
(x-x)^2 + (f(x)-y_d)^2 &= (x-x_f)^2 + (f(x)-y_f)^2 \\
(f(x)-y_d)^2 &= (x-x_f)^2 + (f(x)-y_f)^2 \\
f(x)^2 - 2f(x)y_d + y_d^2 &= (x-x_f)^2 + f(x)^2 - 2f(x)y_f + y_f^2 \\
f(x)^2 - 2f(x)y_d + y_d^2 - f(x)^2 + 2f(x)y_f - y_f^2 &= (x-x_f)^2 \\
2f(x)y_f - 2f(x)y_d + (y_d^2 - y_f^2) &= (x-x_f)^2 \\
2f(x)(y_f - y_d) &= (x-x_f)^2 + (y_f^2 - y_d^2) \\
f(x) &= \frac{(x-x_f)^2}{2(y_f-y_d)} + \frac{y_f^2 - y_d^2}{2(y_f-y_d)} \\
f(x) &= \frac{(x-x_f)^2}{2(y_f-y_d)} + \frac{y_f - y_d}{2(y_f-y_d)} \cdot \frac{y_f + y_d}{2} \\
f(x) &= \frac{(x-x_f)^2}{2(y_f-y_d)} + \frac{(y_f + y_d)}{2}.
\end{aligned}$$

На рисунке ниже показаны:

- Точка $F(x_f, y_f)$ — это фокус параболы.
- Линия $y = y_d$ — директриса, которая всегда лежит ниже фокуса.
- Парабола — это множество точек, равноудалённых от фокуса и директрисы




## 4.1 Основные этапы алгоритма

### 4.1.1 Инициализация очереди событий ($Q$)

- Создать очередь событий $Q$, содержащую все события сайтов (site events) - это точки, где находятся сайты $(p_1, p_2, \ldots, p_n)$.
- Создать пустую структуру статуса $T$ для поддержки текущей конфигурации диаграммы Вороного.
- Создать пустой двусвязный список рёбер $D$ для хранения результата.

### 4.1.2 Сканирующая прямая и ее взаимодействие с диаграммой Вороного

Сканирующая прямая — это концепция, используемая в алгоритме вычисления диаграммы Вороного. Эта прямая проходит горизонтально по плоскости, от верхней части области, содержащей точки-сайты, к нижней части. Основная цель сканирующей прямой заключается в организации обработки событий, связанных с сайтами, и определения структуры диаграммы Вороного в процессе ее движения.

### 4.1.3 Линия пляжа

Линия пляжа (beach line) представляет собой множество парабол, которые определяются каждым сайтом, находящимся выше сканирующей прямой. Для каждого $x$ координаты на линии определяется точка, которая является самой низкой среди всех парабол. Это позволяет гарантировать, что любой вертикальный отрезок ниже линии пересекает её в ровно одной точке, что делает линию $x$-монотонной.

- Линия пляжа изменяется в процессе движения сканирующей прямой. Когда новая парабола появляется, она делит существующую линию пляжа на две части, что может привести к появлению новых разбиений.
- Разбиения (breakpoints) между параболическими арками являются ключевыми точками, которые определяют рёбра диаграммы Вороного. Они отслеживают структурные изменения в диаграмме.

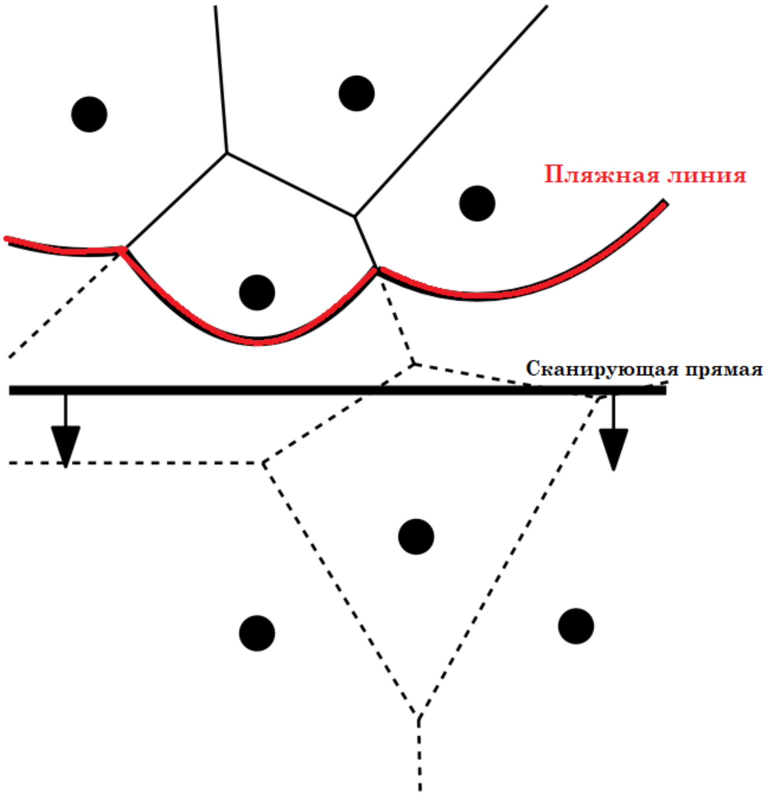



Рис. 2: Построение пляжной линии с помощью сканирующей прямой

### 4.1.4 Понятие события

Когда сканирующая прямая пересекает координаты $y$ определенного сайта $p_i$, происходит *событие сайта* (site event). На этом этапе:

- Учитывается новый сайт $p_i$, и к линии пляжа добавляется новая парабола, соответствующая этому сайту.
- Новая парабола начинает расти вниз по мере движения сканирующей прямой. На начальном этапе она представляет собой вырожденную параболу, которая в дальнейшем расширяется до полной формы.
- В этот момент также могут произойти другие события — например, пересечение новой параболы с уже существующими.

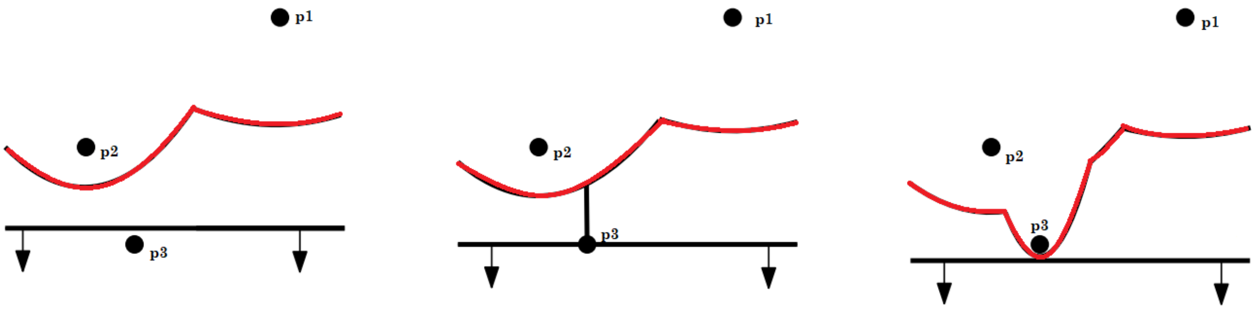



Рис. 3: Обработка добавления новой параболы в beach line

### 4.1.5 Событие окружности

Когда сканирующая прямая продолжает движение вниз и одна из парабол линии пляжа сжимается до точки, это событие называется *событием окружности* (circle event).

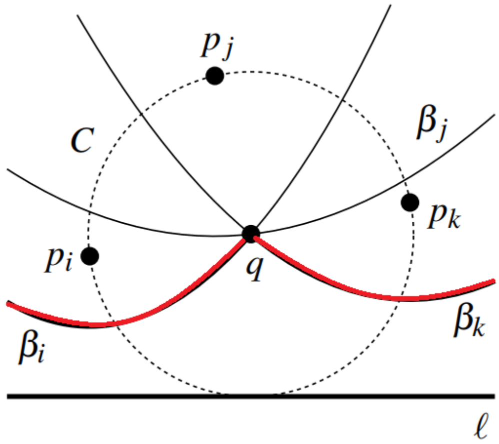



Рис. 4: Наступление события окружности

В этот момент:

- Арка на линии пляжа исчезает, и два других соседних арки соединяются, образуя вершину диаграммы Вороного.
- Вершина представляет собой точку, где расстояния до трех сайтов равны, что делает её основополагающей для определения структуры диаграммы.

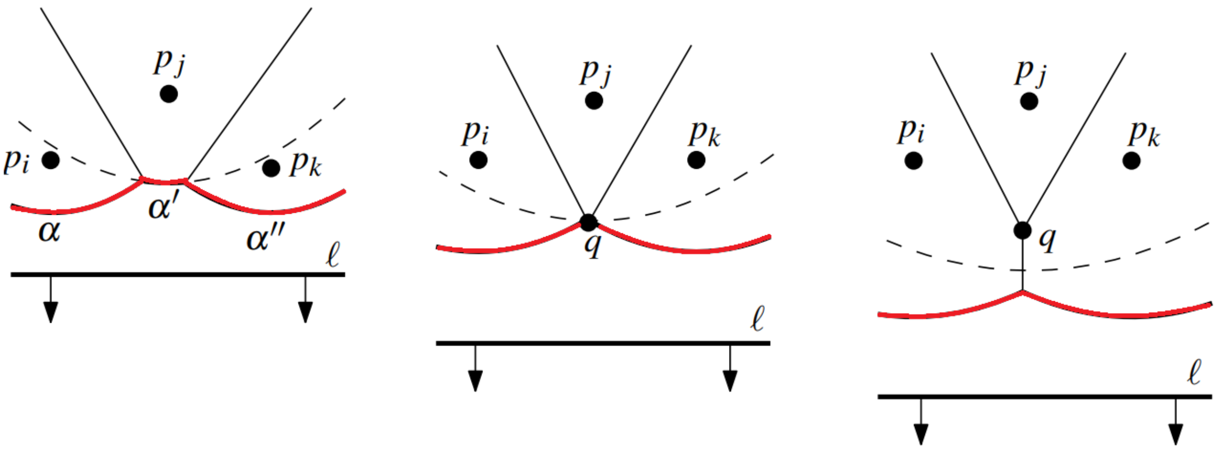



Рис. 5: Продолжение построения после события окружности

### 4.1.6 Обработка оставшихся арков в линии пляжа

- После того как все события в очереди обработаны, линии пляжа могут содержать оставшиеся арки, которые могут соответствовать полубесконечным рёбрам диаграммы Вороного.
- Строить ограничительную рамку, которая охватывает все вершины диаграммы Вороного, и присоединить оставшиеся рёбра к этой рамке.

#### 4.1.7 Завершение

- После обработки всех событий, структура $D$ будет содержать полную диаграмму Вороного, и алгоритм завершён.

На рис.6-11 представлена визуализация нескольких шагов построения диаграммы Вороного.

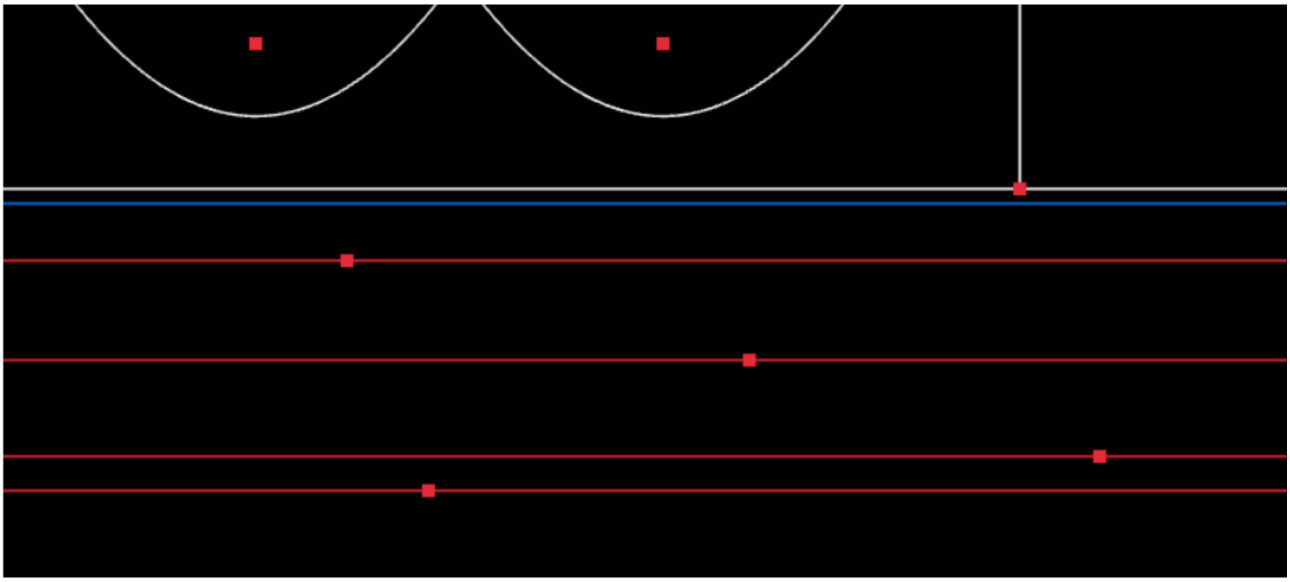


Рис. 6: Визуализация нескольких шагов построения диаграммы Вороного

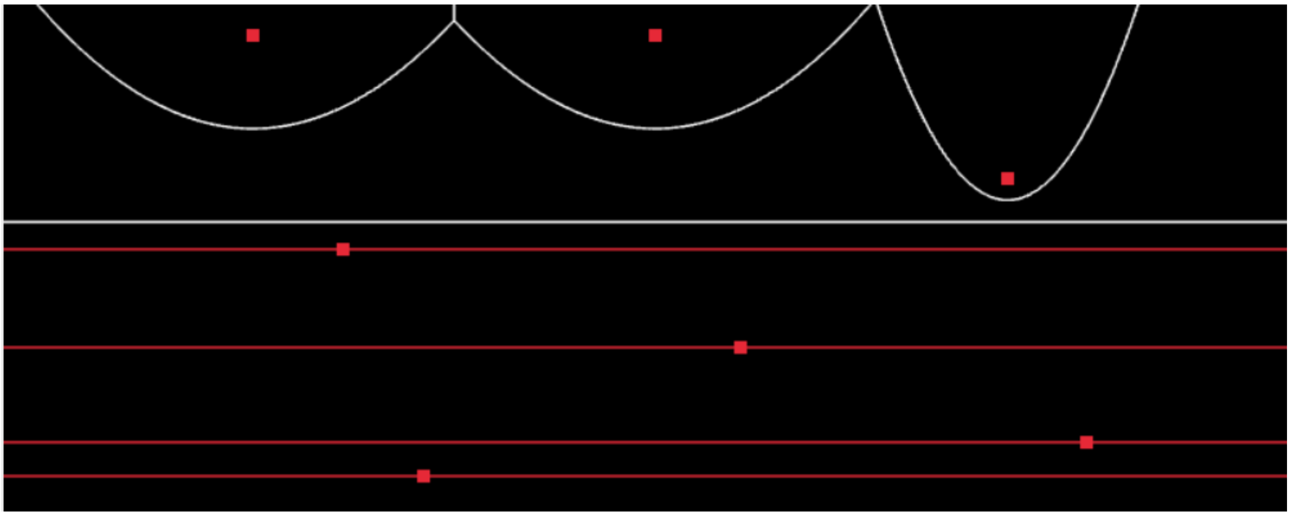


Рис. 7: Визуализация нескольких шагов построения диаграммы Вороного

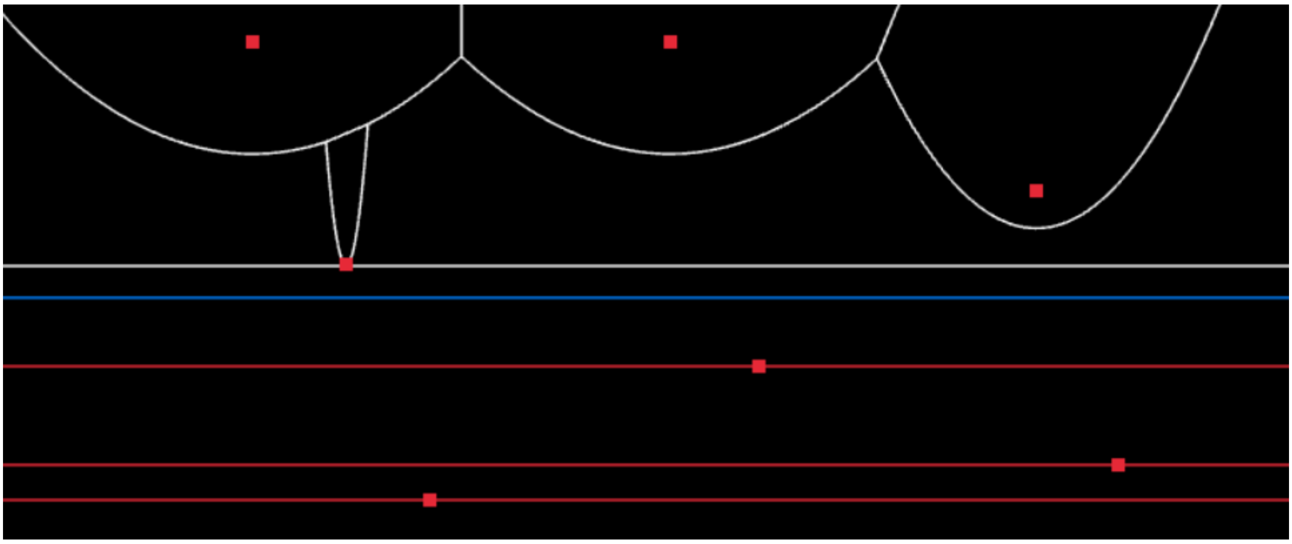


Рис. 8: Визуализация нескольких шагов построения диаграммы Вороного

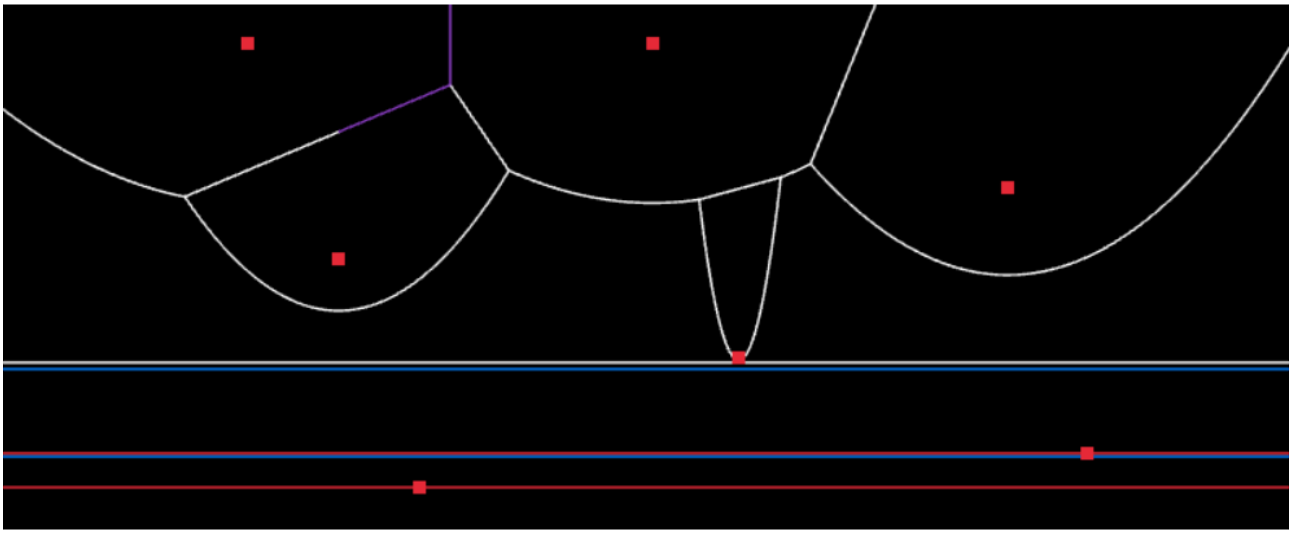


Рис. 9: Визуализация нескольких шагов построения диаграммы Вороного

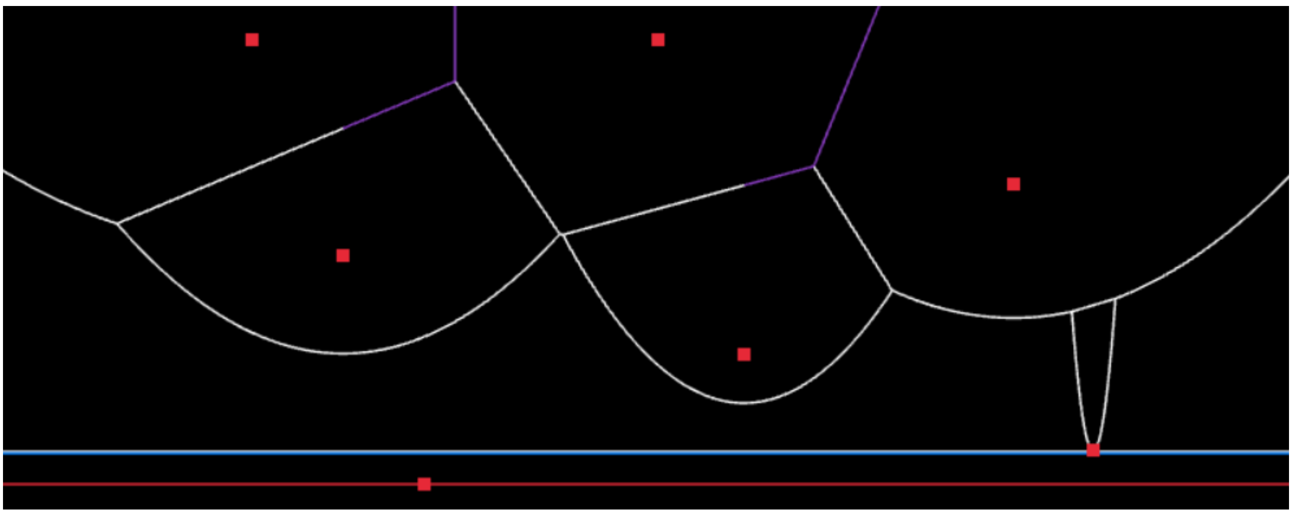


Рис. 10: Визуализация нескольких шагов построения диаграммы Вороного

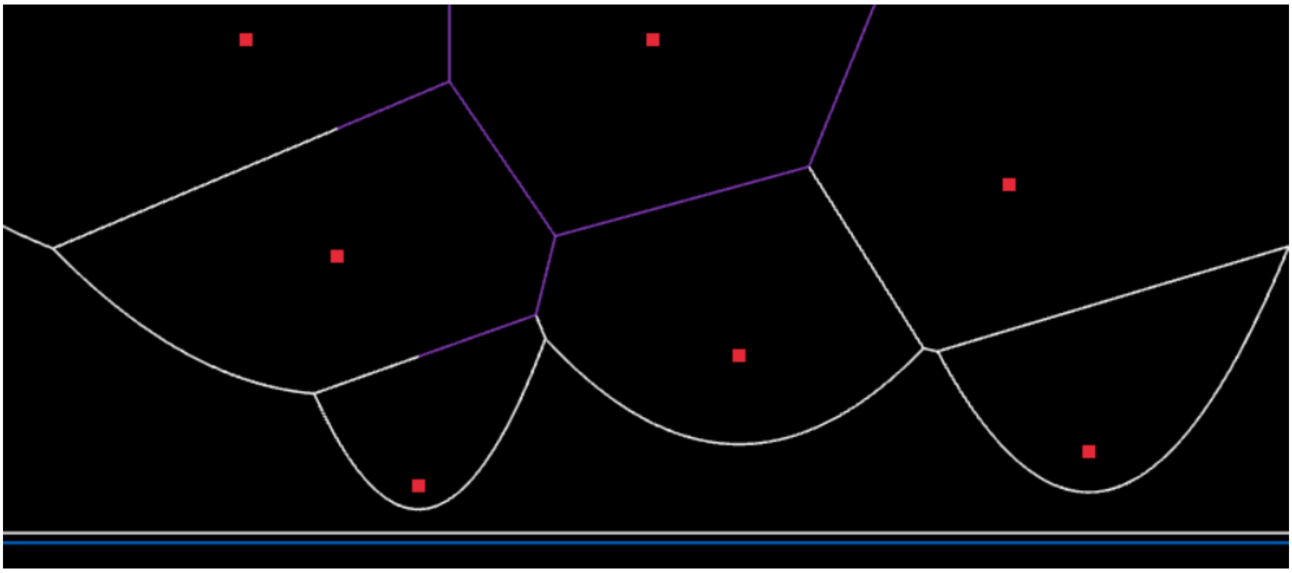


Рис. 11: Визуализация нескольких шагов построения диаграммы Вороного

## 4.2 Блок-схема алгоритма

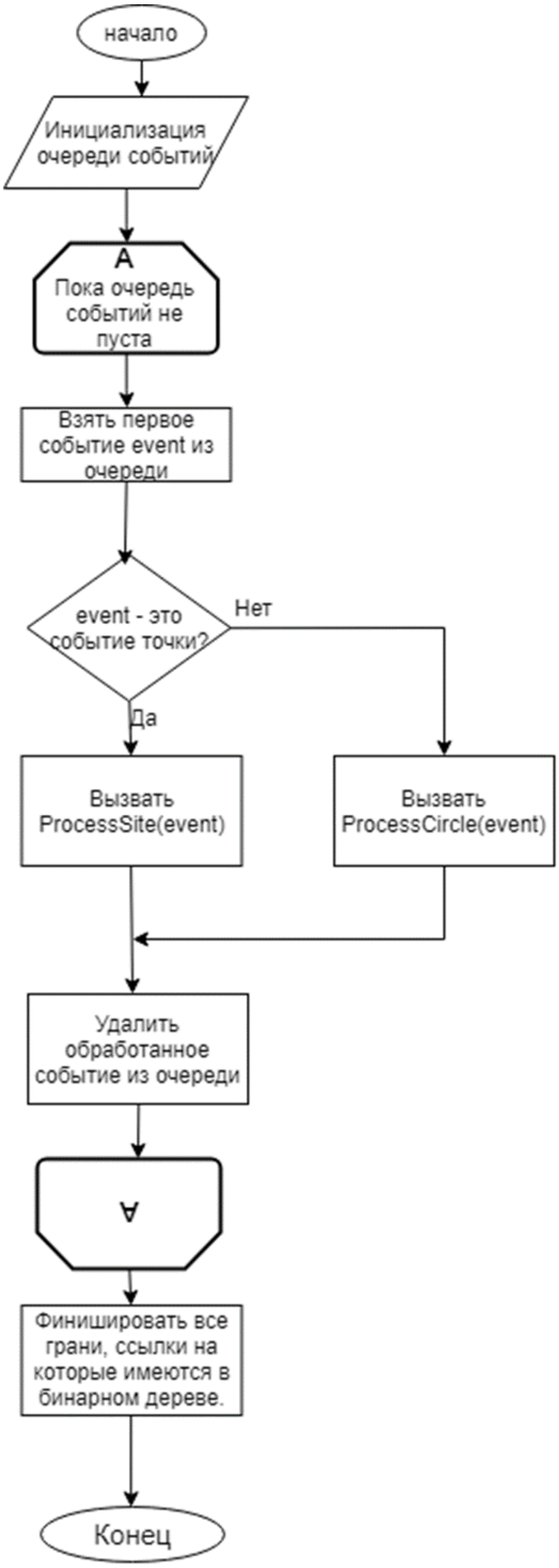



Рис. 12: Блок-схема алгоритма

# 5 Сравнение собственной реализации с библиотечной

Для реализации алгоритма Форчуна был использован Python 3.8 и библиотеки numpy, pyplot. Для сравнения была выбрана библиотека scipy.spatial, в которой, в частности, реализован алгоритм Форчуна для построения диаграммы Вороного.

Для данного исследования была написана функция тестирования, которая измеряет время выполнения реализованной программы и время выполнения построения диаграммы Вороного с помощью функций, реализованных в библиотеке scipy.spatial по следующему алгоритму:

- Запускаем цикл на 20 итераций, внутри цикла:

- Используя библиотеку time, создаём переменную start_time, в которую записываем время запуска тестирования алгоритма.
- Вызывается функция построения диаграммы Вороного VoronoiDiagram() или аналогичная функция Voronoi() в библиотечной реализации.
- В end_time записывается время сразу после выполнения цикла А.
- Далее вычитаем end_time из start_time. Полученная разность записывается в массив.

Далее для 20 значений в массиве после выполнения 20 итераций цикла с помощью библиотеки statistics вычисляется среднее значение времени.

Данное тестирование было проведено для 10, 20, 50, 100 и 500 точек.

| Количество точек | Библиотечная реализация | VoronoiDiagram() | Выигрыш |
|---|---|---|---|
| 10 | 0.0026 сек | 0.0029 сек | -5% |
| 20 | 0.0028 сек | 0.0035 сек | -20% |
| 50 | 0.0035 сек | 0.0043 сек | -19% |
| 100 | 0.0040 сек | 0.0049 сек | -19% |
| 500 | 0.0138 сек | 0.0165 сек | -17% |

Реализация алгоритма Форчуна в сторонней библиотеке имеет меньшее время выполнения, что скорее всего, связано с использованием в библиотечной реализации модуля Qhull для построения диаграммы Вороного. Qhull написан на C, что обеспечивает высокую производительность благодаря компиляции кода.

Моя реализация полностью написана на Python, который интерпретируется, а не компилируется. Это приводит к значительным накладным расходам при работе со сложными структурами данных по сравнению с библиотекой, которая частично написана на C.

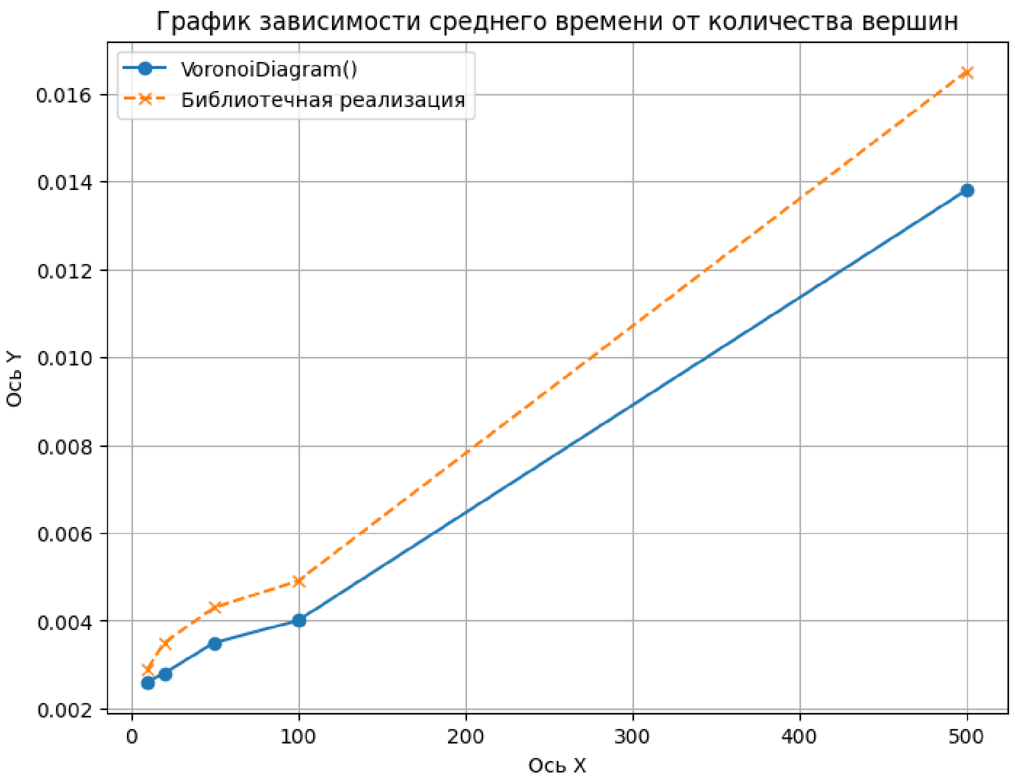



Рис. 13: График зависимости среднего времени от количества вершин

# 6 Заключение

В данной работе были изучены особенности диаграммы Вороного и алгоритмы её построения. В частности, были изучены все аспекты алгоритма Форчуна, а также сделана его реализация на языке Python в среде разработки Visual Studio Code. Кроме того, было произведено сравнение по времени работы реализованной программы с реализацией из сторонней библиотеки. В следствие сравнения, было сделано заключение, что реализованный в собственной программе алгоритм в среднем работает немного медленнее, чем библиотечный. Было сделано предположение, что библиотечная реализация работает быстрее из-за использования низкоуровневых методов библиотеки Qhull.

# Список литературы

[1 ] Препарата Ф., Шеймос М., "Вычислительная геометрия: введение"

# Приложение 1. Код программы

```

class VoronoiDiagram:
    def __init__(self, points, width, height):
        self.point_list = points
        self.reset()
        self.box_x = width
        self.box_y = height

    def reset(self):

        # Инициализация структур данных для нового расчета диаграммы Вороного
        self.event_list = SortedQueue()
        self.beachline_root = None
        self.voronoi_vertex = []
        self.edges = []

    def update(self):

        # Основной метод для построения диаграммы Вороного
        self.reset()
        points = []
        e = None
        for p in self.point_list:
            points.append(Event("point", p))
        self.event_list.points = points

        while self.event_list.length > 0:
            e = self.event_list.extract_first()
            if e.type == "point":
                self.point_event(e.position)
            elif e.active:
                self.circle_event(e)

        self.complete_segments(e.position)

    def point_event(self, p):

        # Обработка события, когда точка попадает в beachline
        q = self.beachline_root
        if q is None:
            self.beachline_root = Arc(None, None, p, None, None)
        else:
            while (q.right is not None and
                    self.parabola_intersection(p.y, q.focus, q.right.focus) <= p.x):
                q = q.right

            e_qp = Edge(q.focus, p, p.x)
            e_pq = Edge(p, q.focus, p.x)

            arc_p = Arc(q, None, p, e_qp, e_pq)
```

```
            arc_qr = Arc(arc_p, q.right, q.focus, e_pq, q.edge["right"])
            if q.right:
                q.right.left = arc_qr
            arc_p.right = arc_qr
            q.right = arc_p
            q.edge["right"] = e_qp

            if q.event:
                q.event.active = False

            self.add_circle_event(p, q)
            self.add_circle_event(p, arc_qr)

            self.edges.append(e_qp)
            self.edges.append(e_pq)

    def circle_event(self, e):

        # Обработка события, когда окружность становится пустой (circle event)
        arc = e.caller
        p = e.position
        edge_new = Edge(arc.left.focus, arc.right.focus)
        if arc.left.event:
            arc.left.event.active = False
        if arc.right.event:
            arc.right.event.active = False

        arc.left.edge["right"] = edge_new
        arc.right.edge["left"] = edge_new
        arc.left.right = arc.right
        arc.right.left = arc.left

        self.edges.append(edge_new)

        if not self.point_outside(e.vertex):
            self.voronoi_vertex.append(e.vertex)

        arc.edge["left"].end = arc.edge["right"].end = edge_new.start = e.vertex

        self.add_circle_event(p, arc.left)
        self.add_circle_event(p, arc.right)

    def add_circle_event(self, p, arc):

        # Добавление circle event в очередь событий
        if arc.left and arc.right:
            a = arc.left.focus
            b = arc.focus
            c = arc.right.focus

            if ((b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y) > 0):
                new_inters = self.edge_intersection(arc.edge["left"], arc.edge["right"])
                circle_radius = ((new_inters.x - arc.focus.x) ** 2 +
                                 (new_inters.y - arc.focus.y) ** 2) ** 0.5
                event_pos = circle_radius + new_inters.y
                if event_pos > p.y and new_inters.y < self.box_y:
```

```
                    e = Event("circle", Point(new_inters.x, event_pos), arc, new_inters)
                    arc.event = e
                    self.event_list.insert(e)

    def parabola_intersection(self, y, f1, f2):

        # Вычисление пересечения двух парабол
        fy_diff = f1.y - f2.y
        if fy_diff == 0:
            return (f1.x + f2.x) / 2

        fx_diff = f1.x - f2.x
        b1md = f1.y - y
        b2md = f2.y - y
        h1 = (-f1.x * b2md + f2.x * b1md) / fy_diff
        h2 = (b1md * b2md * (fx_diff ** 2 + fy_diff ** 2)) ** 0.5 / fy_diff

        return h1 + h2

    def edge_intersection(self, e1, e2):

        # Вычисление пересечения двух ребер
        if e1.m == float('inf'):
            return Point(e1.start.x, e2.get_y(e1.start.x))
        elif e2.m == float('inf'):
            return Point(e2.start.x, e1.get_y(e2.start.x))
        else:
            mdif = e1.m - e2.m
            if mdif == 0:
                return None
            x = (e2.q - e1.q) / mdif
            y = e1.get_y(x)
            return Point(x, y)

    def complete_segments(self, last):

        # Завершение ребер, которые не были завершены из-за событий
        r = self.beachline_root
        while r.right:
            e = r.edge["right"]
            x = self.parabola_intersection(last.y * 1.1, e.arc["left"], e.arc["right"])
            y = e.get_y(x)
            if ((e.start.y < 0 and y < e.start.y) or
                (e.start.x < 0 and x < e.start.x) or
                (e.start.x > self.box_x and x > e.start.x)):
                e.end = e.start
            else:
                if e.m == 0:
                    x = 0 if x - e.start.x <= 0 else self.box_x
                    e.end = Point(x, e.start.y)
                    self.voronoi_vertex.append(e.end)
                else:
                    if e.m == float('inf'):
                        y = self.box_y
                    else:
                        y = 0 if e.m * (x - e.start.x) <= 0 else self.box_y
```

```
                    e.end = self.edge_end(e, y)
            r = r.right

        for i in range(len(self.edges)):
            e = self.edges[i]
            if e is None:
                continue

            option = 1 * self.point_outside(e.start) + 2 * self.point_
               outside(e.end)

            if option == 3:
                self.edges[i] = None
            elif option == 1:
                y = 0 if e.start.y < e.end.y else self.box_y
                e.start = self.edge_end(e, y)
            elif option == 2:
                y = 0 if e.end.y <= e.start.y else self.box_y
                e.end = self.edge_end(e, y)

    def edge_end(self, e, y_lim):

        # Вычисление конца ребра на границе области
        x = min(self.box_x, max(0, e.get_x(y_lim)))
        y = e.get_y(x)
        if y is None:
            y = y_lim
        p = Point(x, y)
        self.voronoi_vertex.append(p)
        return p

    def point_outside(self, p):

        # Проверка, находится ли точка за пределами области
        return p.x < 0 or p.x > self.box_x or p.y < 0 or p.y > self.box_y


class Arc:
    def __init__(self, l, r, f, el, er):

        # Дуга параболы в beachline
        self.left = l
        self.right = r
        self.focus = f  # Point
        self.edge = {"left": el, "right": er}  # Edge
        self.event = None


class Point:
    def __init__(self, x, y):

        # Простая точка
        self.x = x
        self.y = y


class Edge:
    def __init__(self, p1, p2, startx=None):

        # Ребро диаграммы Вороного
```

```
        if p1.y - p2.y == 0:
            self.m = float('inf')
        else:
            self.m = -(p1.x - p2.x) / (p1.y - p2.y)

        if p1.y - p2.y == 0:
            self.q = None
        else:
            self.q = (0.5 * (p1.x ** 2 - p2.x ** 2 + p1.y ** 2 - p2.y **
                2)) / (p1.y - p2.y)

        self.arc = {"left": p1, "right": p2}
        self.end = None
        self.start = None

        if startx is not None:
            self.start = Point(startx, None if self.m == float('inf')
                else self.get_y(startx))

    def get_y(self, x):

        # Вычисление y координаты на ребре для заданного x
        if self.m == float('inf'):
            return None
        return x * self.m + self.q

    def get_x(self, y):

        # Вычисление x координаты на ребре для заданного y
        if self.m == float('inf'):
            return self.start.x
        return (y - self.q) / self.m


class Event:
    def __init__(self, type, position, caller=None, vertex=None):

        # Событие в очереди событий (point event или circle event)
        self.type = type
        self.caller = caller
        self.position = position
        self.vertex = vertex
        self.active = True


class SortedQueue:
    def __init__(self, events=None):

        # Очередь событий, отсортированная по координате y
        self.list = []
        if events:
            self.list = events
        self.sort()

    @property
    def length(self):
        return len(self.list)

    def extract_first(self):

```

```
        # Извлечение первого события из очереди
        if len(self.list) > 0:
            elm = self.list[0]
            self.list.pop(0)
            return elm
        return None

    def insert(self, event):

        # Вставка события в очередь
        self.list.append(event)
        self.sort()

    @property
    def points(self):
        return self.list

    @points.setter
    def points(self, events):
        self.list = events
        self.sort()

    def sort(self):

        # Сортировка очереди событий
        self.list.sort(key=lambda a: (a.position.y, a.position.x))
```