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 *Робототехники и комплексной автоматизации*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

ОТЧЕТ О ВЫПОЛНЕНИИ ДОМАШНЕГО ЗАДАНИЯ №1

по курсу: «Автоматизация технологического проектирования»

Студент	Жидков Антон Алексеевич
Группа	РК6-31М
Тип задания	Домашнее задание

Студент _____ **Жидков А.А.**
подпись, дата *фамилия, и.о.*

Преподаватель _____ **Божко А.Н.**
подпись, дата *фамилия, и.о.*

Москва, 2024 г.

Оглавление

1. Задание

1. Разработать алгоритм, решающий задачу планирования перемещений триангуляцией свободного пространства и поиска кратчайшего пути.

2. Введение

Задача планирования перемещений робота в конфигурационном пространстве представляет собой важную область исследований в робототехнике и автоматизации. Основная цель данной задачи заключается в разработке алгоритмов и методов, которые позволяют роботу, эффективно и безопасно перемещаться в пространстве, избегая препятствий и достигая заданных целей.

В данной задаче робот рассматривается как точка, которая может перемещаться по плоскости. Пространство, в котором движется робот, представляет собой плоскую поверхность, на которой могут находиться различные препятствия. Эти препятствия являются статическими, и робот должен учитывать их положение при планировании своего маршрута.

Основные концепции, связанные с планированием перемещений робота, включают в себя представление пространства, моделирование препятствий, разработку алгоритмов поиска пути и оптимизацию маршрутов. Представление пространства может включать использование сеток, графов или других структур данных, которые позволяют эффективно хранить и обрабатывать информацию о пространстве и препятствиях. Моделирование препятствий включает в себя определение их формы, размера и положения, а также прогнозирование их возможных перемещений, если они динамические.

Алгоритмы поиска пути играют ключевую роль в планировании перемещений робота. Эти алгоритмы должны быть способны находить оптимальные или приемлемые маршруты от начальной точки до целевой, учитывая наличие препятствий и другие ограничения. Оптимизация маршрутов направлена на минимизацию времени, затраченного на перемещение, энергопотребления или других критериев, важных для конкретной задачи.

В данной работе робот представляется точкой, все препятствия являются стационарными, их положение известно. Робот не имеет

органов чувств. Все элементы робота твёрдые тела, на траекторию движения робота не накладывается ограничений. Вся информация о внешней среде известна.

Путь является допустимым, если он не пересекает не пересекает препятствия. Пример допустимого и недопустимого путей представлен на рисунке 1.

а) б)

Рисунок 1 – Пример а) недопустимого пути, б) допустимого пути.

Рассматриваемый алгоритм решения задачи планирования перемещения относится к графовым офлайн методам. В данном случае конфигурационное пространство заранее известно. Пространство делится на треугольники и покрывается точками, соответствующими характерным точкам. В каждой точке роботу разрешается перемещение к соседним точкам до тех пор, пока линия между ними полностью находится в пределах C_{free}. Данный подход позволяет использовать для поиска пути алгоритмы поиска (алгоритм Дейкстры, A*). На рисунке 2 представлен пример работы графового метода планирования перемещения.

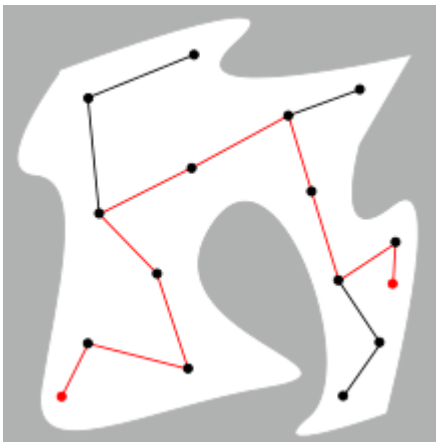


Рисунок 2 – Графовый метод планирования перемещения.

3. Алгоритмы

3.1. Триангуляция Делоне

Триангуляцией [1] называется планарное разбиение плоскости на плоские фигуры, из которых одна является внешней бесконечностью, а остальные – треугольниками. Будем рассматривать задачу построения триангуляции по заданному набору S двумерных точек. Эта задача состоит в соединении заданных точек из S прямыми отрезками так, чтобы никакие отрезки не пересекались. Решение этой задачи неоднозначно, поэтому возникает проблема построения оптимальной триангуляции. Оптимальной называют такую триангуляцию, у которой сумма длин всех ребер минимальна, однако построение такой триангуляции имеет сложность $O(e^N)$[2]. Это ограничивает применение алгоритмов построения оптимальной триангуляции на практике.

Рассмотрим триангуляцию Делоне. Триангуляция Делоне – это такая триангуляция, при которой ни одна из точек набора S не попадает внутрь ни одной из описанных вокруг полученных треугольников окружностей за исключением точек, являющихся его вершинами.

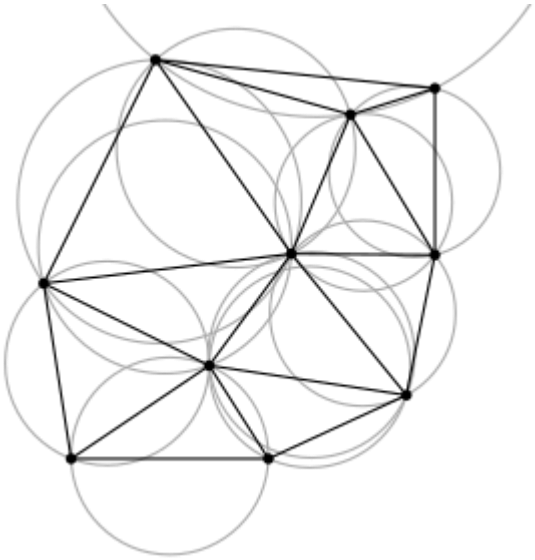


Рисунок 3 – пример триангуляции Делоне

Триангуляция Делоне не является оптимальной, но она строит набор треугольников, которые «стремятся к равноугольности» и имеет некоторые важные свойства: триангуляция Делоне обладает

максимальной суммой минимальных углов всех своих треугольников среди всех возможных триангуляций на заданном наборе точек; триангуляция Делоне обладает минимальной суммой радиусов окружностей, описанных около треугольников, среди всех возможных триангуляций на заданном наборе точек.

Описание итеративного алгоритма Делоне.

Итеративный алгоритм предполагает пошаговое создание триангуляции Делоне, где на каждом шаге добавляется новый треугольник, соответствующий критерию Делоне, к уже существующей структуре. При добавлении точки могут возникнуть четыре основных ситуации:

- Точка находится внутри одного из существующих треугольников триангуляции.
- Точка расположена за пределами текущей триангуляции.
- Точка лежит на ребре одного из треугольников.
- Точка совпадает с одной из вершин треугольников триангуляции.

В случае, если точка попадает внутрь треугольника, последний разделяется на несколько новых, с обязательной проверкой условия Делоне. Если точка находится за пределами триангуляции, добавляются новые внешние треугольники с последующей проверкой на соответствие условию. Когда точка оказывается на ребре, происходит разделение ребра и связанных с ним треугольников. Если же точка совпадает с одной из существующих вершин, её добавление игнорируется.

Общая сложность алгоритма складывается из этапов поиска подходящего треугольника для добавления точки, создания новых треугольников, проверки выполнения условия Делоне и перестроения триангуляции при необходимости. Поиск треугольника оценивается как $O(N)$, тогда как построение новых треугольников и проверка их соответствия обычно требуют фиксированного числа операций.

Ситуацию, когда точка находится вне триангуляции, можно избежать, если начать процесс с большого стартового треугольника, охватывающего все точки. При добавлении каждой новой точки может потребоваться перестроение значительной части триангуляции, что приводит к общей вычислительной сложности алгоритма порядка $O(N^2)$[1].

3.2. Алгоритм Боуэра-Ватсона

Алгоритм Боуэра-Ватсона [3], [4] – инкрементный алгоритм, который позволяет получить триангуляцию Делоне для конечного набора точек. Для создания триангуляции Делоне задаётся супер-треугольник, который охватывает все заданные точки. Далее точки добавляются к действительной триангуляции Делоне по одной. На рисунке 1 изображен супер-треугольник с первой добавленной точкой.

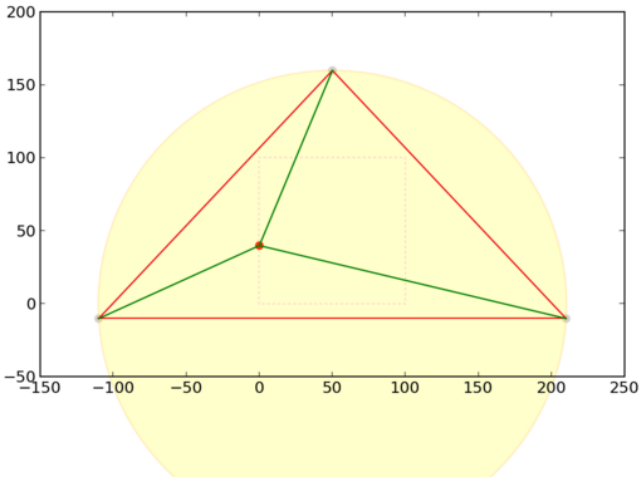


Рисунок 4 – Вставка точки в супер-треугольник

После добавление точки происходит проверка условия непринадлежности точки к описанной окружности каждого треугольника триангуляции. На рисунке 4 видно, что описанная вокруг супер-треугольника окружность содержит добавленную точку. В таком случае нужно перестроить триангуляцию. Для этого все треугольники, описанные окружности которых содержат точку, образуют границы многоугольной пустоты, после чего данные треугольники удаляются из

триангуляции. Затем происходит процесс триангуляции многоугольной пустоты и добавление её к исходной триангуляции. Результат данного этапа представлен на рисунке 5.

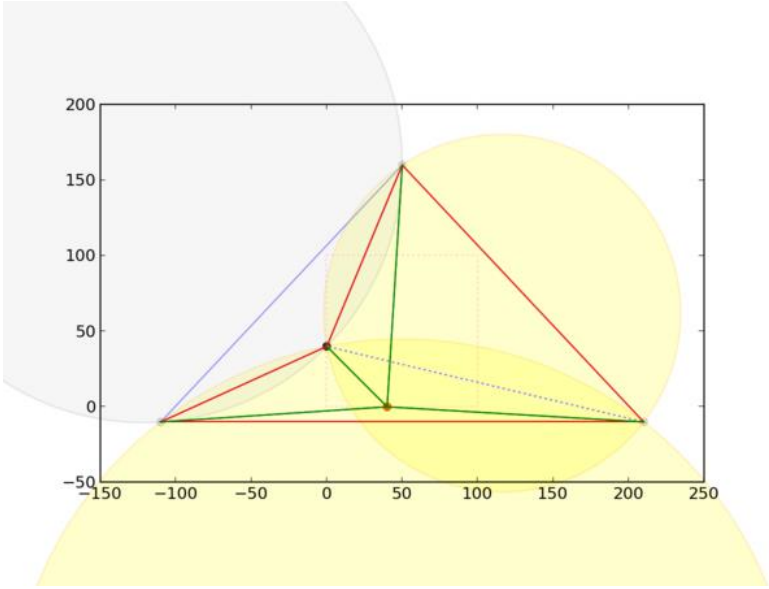


Рисунок 5 – Результат перестроения триангуляции и добавление ещё одной точки.

На рисунке 5 визуально разделены окружности, описанные вокруг треугольников, содержащие точку, не являющейся вершиной треугольника, (желтым цветом) и не содержащие подобных точек (серые). После добавления всех точек получим триангуляцию внутри супер-треугольника (рисунок 6). После удаления супер-треугольника будет найдена искомая триангуляция Делоне.

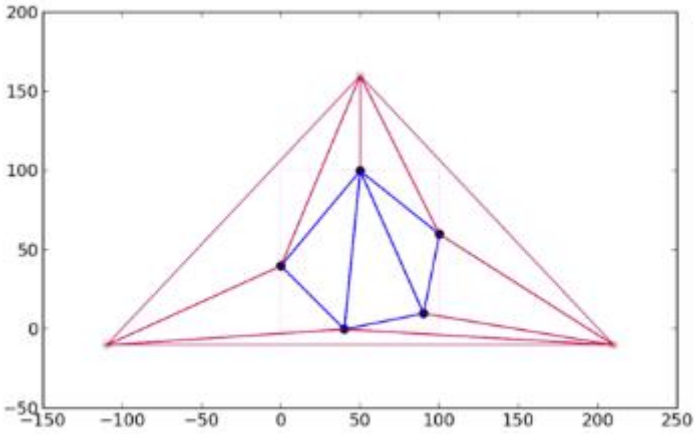


Рисунок 6 – Триангуляция внутри супер-треугольника.

3.3. Алгоритм Уайлера-Атертона

Алгоритм Уайлера-Атертона [5] позволяет найти область пересечения отсекаемого многоугольника по отсекающему многоугольнику. Отсекаемый и отсекающий многоугольники могут быть невыпуклыми. Алгоритм является итерационным и описывается как:

1. Составление списка из координат вершин A и B с сортировкой вершин по часовой стрелке для полигона A и B.
2. Определение точек пересечения A и B при проходе полигона по часовой стрелке с пометкой о том, входит ли полигон B в полигон A в этой точке или выходит.

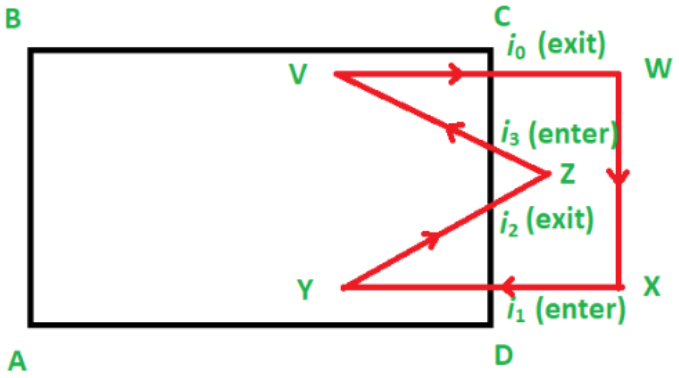


Рисунок 7 – Определение точек пересечения при обходе полигона VWXY по часовой стрелке

3. Создание и заполнение двух списков обхода полигонов A и B с добавлением точек пересечения многоугольников.

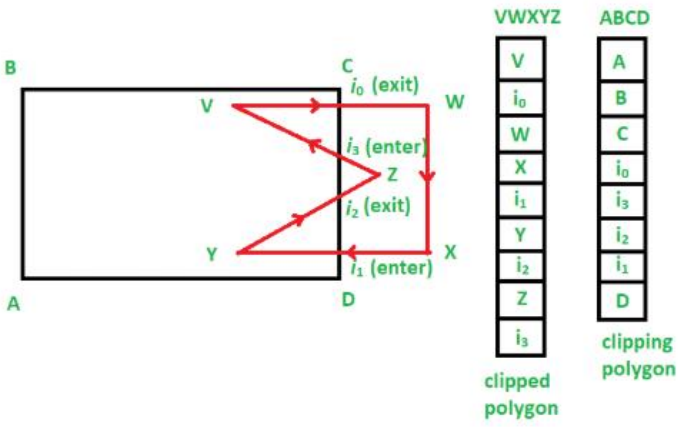


Рисунок 8 – Составление списков обхода полигонов

4. Обход полигона по первому списку, поиск первой входящей точке пересечения. При нахождении входящей точки пересечения, обход продолжается, начиная с этой точки во втором списке. При нахождении следующей точки пересечения образуется замкнутый контур из рассмотренных точек, который является один из наложений препятствий.

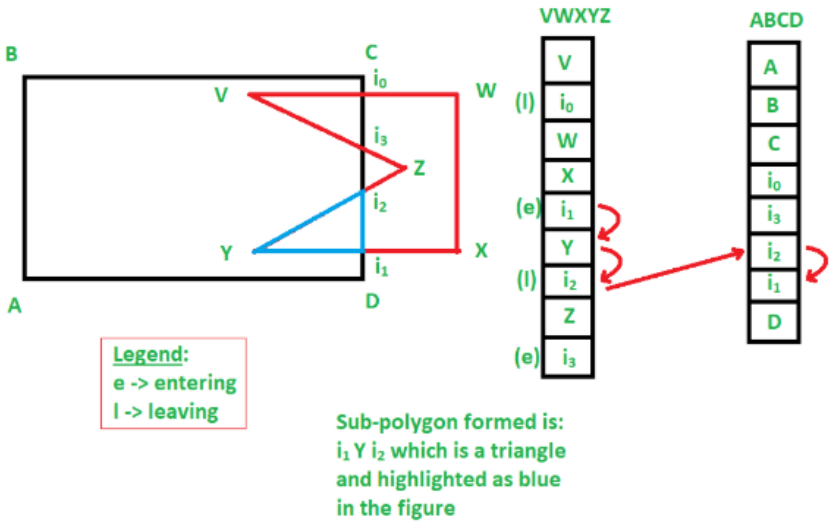


Рисунок 9 - Нахождение контура наложения препятствий.

5. После нахождения всех контуров наложения полигонов производится операция отсечения полученных контуров из первого полигона.
6. Если ни одного пересечения не найдено, возникает одна из следующих ситуаций:
    a. A внутри B — вернуть A при отсечении, B при объединении.
    b. B внутри A — вернуть B при отсечении, A при объединении.
    c. A и B не пересекаются — вернуть пустое множество при отсечении, A&B при объединении.

3.4. Составление графа и поиск кратчайшего пути

Для составления графа разработан алгоритм прохождения по всем треугольникам полученной триангуляции и добавления характерных точек треугольника в итоговый граф. В качестве характерных точек

выбраны центр окружности, описанной вокруг треугольника и середины сторон треугольника. Перед каждым добавлением точки в граф производится проверка принадлежности этой точки к препятствию.

При триангуляции поля с многогранными препятствиями могут возникать случаи, когда рассмотренные характерные точки не обеспечивают составление графа. Данные случаи встречаются при такой триангуляции, когда в треугольник попадает часть препятствия. В этом случае предлагается добавить в граф середины отрезков, не лежащих на препятствии в случае, если середина стороны треугольника принадлежит препятствию. Для этого в алгоритм добавлен поиск точек пересечения препятствий и добавления середин соответствующих отрезков.

Рёбра графа соответствуют связям добавляемых вершин внутри треугольника по принципу «все со всеми». Перед добавлением связи выполняется проверка на достижимость из одной точки в другую, то есть не пересекает ли путь препятствие.

Поиск кратчайшего пути выполнен при помощи алгоритма Дейкстры, где в качестве веса ребра используется евклидово расстояние.

4. Реализация

4.1. Основные сущности и вспомогательные функции

Для построения заданного пространства было реализовано считывание начальной и целевой точек и препятствий из генерируемого файла формата .json.

```
def filter_by_type(data, type_name):
    return [item for item in data if item.get('type') == type_name]

info_elements = filter_by_type(data, 'info')
start_points = filter_by_type(data, 'startPoint')
end_points = filter_by_type(data, 'endPoint')
polygons = filter_by_type(data, 'polygon')
```

Был описан ряд сущностей для реализации алгоритма.

Класс точки:

```
class Point:
    def __init__(self, x=None, y=None, dict=None) -> None:
        if dict is not None:
            self.x = dict[0].get('x')
            self.y = dict[0].get('y')
        else:
            self.x = x
            self.y = y

    def __eq__(self, other: object) -> bool:
        if isinstance(other, Point):
            return self.x == other.x and self.y == other.y
        return False
```

Класс препятствия

```
class Obstacle:
    def __init__(self, points: list[Point], ensure = True) -> None:
        if ensure:
            self.points = self.ensure_clockwise(points)
        else:
            self.points = points
        self.color = (random.random(), random.random(), random.random())
```

Для дальнейшей реализации алгоритма слияния препятствий при создании препятствия происходит упорядочение вершин по часовой стрелке:

```
def ensure_clockwise(points: list[Point]) -> list[Point]:
        """
        Упорядочивает точки многоугольника по часовой стрелке.
        """
        # Найти центр многоугольника
        center_x = sum(p.x for p in points) / len(points)
        center_y = sum(p.y for p in points) / len(points)

        # Сортировать точки по углу относительно центра
        def angle(p):
```

```
            return atan2(p.y - center_y, p.x - center_x)

        sorted_points = sorted(points, key=angle)
        return sorted_points
```

Класс треугольника

```
class Triangle:
    def __init__(self, p1, p2, p3):
        self.points = [p1, p2, p3]
        self.edges = [(p1, p2), (p1, p3), (p2,p3)]
        while self.ccw():
            self.points = [p1, p3, p2]
```

Класс поля

```
class Field:
    def __init__(self, start: Point, finish: Point, edges: list[Point],
obstacles: list[Obstacle] = None, plot = False) -> None:
        self.start_point = start
        self.finish_point = finish
        self.edges = edges
        self.obstacles = obstacles
        self.points: list[Point] = []
        self.plot = plot
```

4.2. Алгоритм Боуэра-Вотсона

Реализацию алгоритма можно разбить на несколько шагов. Согласно алгоритму, начальная триангуляция определяется супер-треугольником. Для этого определяются максимальные и минимальные значения координат и задаётся треугольник, гарантированно покрывающий всё пространство триангуляции.

```
    # Построение супер треугольника
    min_x = min(point.x for point in points)
    min_y = min(point.y for point in points)
    max_x = max(point.x for point in points)
    max_y = max(point.y for point in points)
```

```
range_x = max_x - min_x
range_y = max_y - min_y

max_range = max(range_x, range_y)

center_x = (min_x + max_x) / 2
center_y = (min_y + max_y) / 2

p1 = Point(center_x - 20 * max_range, center_y - max_range)
p2 = Point(center_x, center_y + 20 * max_range)
p3 = Point(center_x + 20 * max_range, center_y - max_range)

super_triangle = Triangle(p1, p2, p3)

# Инициализация триангуляции супер-треугольником
triangles = [super_triangle]
```

После чего происходит поочерёдное добавление точек. При добавлении каждой точки определяются те треугольники, описанная окружность которых содержит добавленную точку. Эти треугольники отмечаются как плохие. Далее из этих треугольников составляется многоугольная пустота, плохие треугольники удаляются из триангуляции и проводится перетриангуляция пустоты. После добавления всех точек происходит удаление супер-треугольника.

```
for point in points:
    bad_triangles = []
    # Проверка находится ли точка в окружности треугольника
    for triangle in triangles:
        if triangle.in_circumcircle(point):
            bad_triangles.append(triangle)
    polygon = []
    # Формирование многоугольной пустоты
    for triangle in bad_triangles:
        not_triangle = [x for x in bad_triangles if x != triangle]
        for edge in triangle.edges:
            shared_count = 0
            for other_triangle in not_triangle:
```

```
                for other_edge in other_triangle.edges:
                    if set(edge) == set(other_edge):
                        shared_count += 1
            if shared_count == 0:
                polygon.append(edge)
    # Удаление плохих треугольников
    for triangle in bad_triangles:
        triangles.remove(triangle)
    # Добавление новых треугольников в триангуляцию
    for edge in polygon:
        triangle = Triangle(edge[0], edge[1], point)
        triangles.append(triangle)

# Удаление супер-треугольника
triangles_to_remove = []
for i in range(len(triangles)):
    triangle = triangles[i]
    for point in triangle.points:
        if point in super_triangle.points:
            triangles_to_remove.append(i)

for index in sorted(list(set(triangles_to_remove)), reverse=True):
    del triangles[index]
return triangles
```

4.3. Алгоритм Уайлера-Атертона

Для реализации алгоритма был разработан ряд вспомогательных методов. Для поиска пересечения двух прямых используется функция определения расположения точки слева от отрезка. Если начальные и конечные точки отрезков находятся по разные стороны относительно сравниваемого отрезка, то эти отрезки пересекаются. Данная проверка реализована как:

$$\text{ccw}(A, B, C) = (C.y - A.y) \cdot (B.x - A.x) > (B.y - A.y) \cdot (C.x - A.x)$$

```
    def ccw(A, B, C):
        return (C.y - A.y) * (B.x - A.x) > (B.y - A.y) * (C.x - A.x)

    def intersect(A, B, C, D):
```

```
            return ccw(A, C, D) != ccw(B, C, D) and ccw(A, B, C) != ccw(A, B,
D)
```

Если точки пересекаются, то точку их пересечения можно найти при помощи определителя:

```
        if intersect(p1, p2, p3, p4):
            # Найти точку пересечения
            xdiff = (p1.x - p2.x, p3.x - p4.x)
            ydiff = (p1.y - p2.y, p3.y - p4.y)

            def det(a, b):
                return a[0] * b[1] - a[1] * b[0]

            div = det(xdiff, ydiff)
            if div == 0:
                return None

            d = (det(*[(p1.x, p1.y), (p2.x, p2.y)]), det(*[(p3.x, p3.y),
(p4.x, p4.y)]))
            x = det(d, xdiff) / div
            y = det(d, ydiff) / div
            return Point(x, y)
        return None
```

Согласно описанию алгоритма, необходимо найти все точки пересечения препятствий и сформировать два списка обхода препятствий по часовой стрелке с учётом точек пересечения. Поскольку изначально при задании препятствия вершины сортируются в порядке часовой стрелки, необходимо только добавить точки пересечения на соответствующие отрезки.

```
    def find_intersections(self, other: 'Obstacle') -> list[Point]:
        """
        Находит точки пересечения двух препятствий.
        """
        intersections = []
        for i in range(len(self.points)):
            p1 = self.points[i]
```

```
            p2 = self.points[(i + 1) % len(self.points)]
            for j in range(len(other.points)):
                p3 = other.points[j]
                p4 = other.points[(j + 1) % len(other.points)]
                intersection = Obstacle.line_intersection(p1, p2, p3, p4)
                if intersection:
                    intersections.append(intersection)
        return intersections


    def define_arrays(self, other: 'Obstacle', intersections: list[Point]):
        """
        Формирует два списка точек с учетом сортировки по часовой стрелке и
добавления точек пересечения.
        """
        def insert_intersections(points, intersections):
            result = []
            for i in range(len(points)):
                current_point = points[i]
                next_point = points[(i + 1) % len(points)]

                # Добавляем текущую вершину
                result.append(current_point)

                segment_intersections = [
                    p for p in intersections if is_point_on_segment(p,
current_point, next_point)
                ]

                segment_intersections.sort(key=lambda p:
distance_squared(current_point, p))

                result.extend(segment_intersections)

            return result

        self_points = insert_intersections(self.points, intersections)
        other_points = insert_intersections(other.points, intersections)

        return self_points, other_points
```

Если для двух препятствий не было найдено точек пересечения, необходимо рассмотреть случай вложенности одного препятствия в другое. Если препятствия не вложены, значит они не пересекаются. В ином случае необходимо найти первую точку препятствия, находящуюся вне другого препятствия. Таким образом будет задано начало обхода.

Сам обход аналогичен описанному ранее алгоритму за исключением, что реализуется объединение препятствий. Таким образом, если при обходе встретилась точка пересечения препятствий, обход продолжается со следующей точки другого списка. Обход продолжается до тех пор, пока не достигнута начальная точка.

```
    def merge_with(self, other: 'Obstacle') -> 'Obstacle' or
tuple['Obstacle', 'Obstacle']:
        """
        Объединяет два препятствия в одно, если они пересекаются или одно
вложено в другое, иначе возвращает два препятствия.
        """
        intersections = self.find_intersections(other)
        if not intersections:
            if self.is_point_inside(self.points[0], other.points):
                return other
            elif self.is_point_inside(other.points[0], self.points):
                return self
            # Если нет пересечений, возвращаем два препятствия
            return self, other
        self_points, other_points = self.define_arrays(other, intersections)

        merged_points = []
        idx = 0
        first_outher_idx = -1
        for point in self_points:
            if point in self.points:
                # определи находится ли эта точка внутри препятствия other
                if not Obstacle.is_point_inside(point, other.points):
                    # нашли первую точку снаружи второго препятствия, Значит
можно определить первую точку выхода
                    if idx != 0:
                        first_outher_idx = idx - 1
```

```
                    break
            idx += 1

        # теперь надо начать с начала списка (если first_outher_idx == -1)
или с перой точки выхода
        idx_start = idx
        if first_outher_idx != -1:
            idx_start = first_outher_idx
        self_flag = True
        merged_points.append(self_points[idx_start])
        current_idx = idx_start + 1
        while True:
            if self_flag:
                if self_points[current_idx] in self.points:
                    merged_points.append(self_points[current_idx])
                    current_idx = (current_idx + 1) % len(self_points)
                else:
                    merged_points.append(self_points[current_idx])
                    current_point = self_points[current_idx]
                    other_idx = other_points.index(current_point)
                    self_flag = False
                    current_idx = (other_idx + 1) % len(other_points)
            else:
                if other_points[current_idx] in other.points:
                    merged_points.append(other_points[current_idx])
                    current_idx = (current_idx + 1) % len(other_points)
                else:
                    merged_points.append(other_points[current_idx])
                    current_point = other_points[current_idx]
                    self_idx = self_points.index(current_point)
                    self_flag = True
                    current_idx = (self_idx + 1) % len(self_points)

            if merged_points[0] == merged_points[-1]:
                break

        if merged_points[0] == merged_points[-1]:
            merged_points.pop()

        return Obstacle(merged_points, ensure=False)
```

Демонстрация работы алгоритма.

На рисунке 10 изображено заданное пространство, на рисунке 11 – поочередное слияние препятствий и результат.

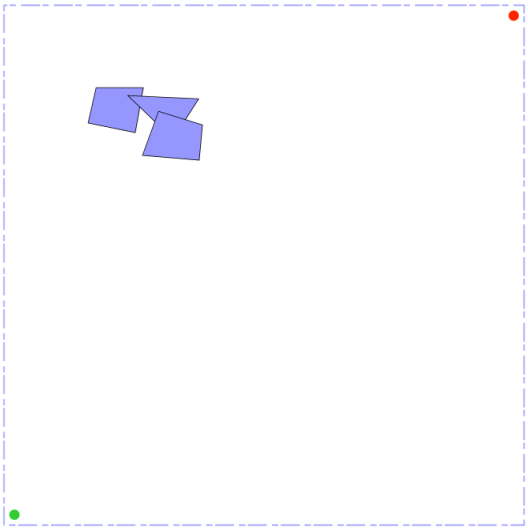


Рисунок 10 – Заданное пространство с пересекающимися препятствиями.

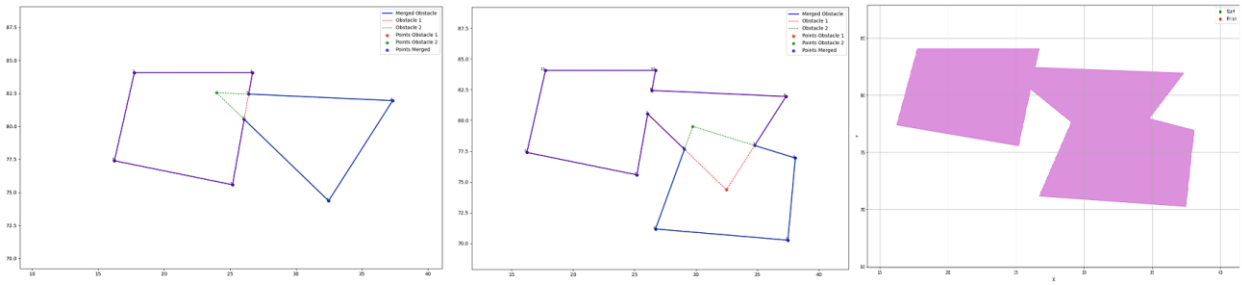


Рисунок 11 – Процесс слияния препятствий и результат слияния.

4.4. Составление графа и поиск кратчайшего пути

Для составления графа и поиска кратчайшего пути используется библиотека networkx. Для каждого треугольника в полученной триангуляции выбираются характерные точки: центр описанной вокруг треугольника окружности и середины сторон треугольника. Для каждой стороны проверяется, принадлежит ли её середина какому-либо препятствию. В случае, если точка не принадлежит препятствию, она добавляется в граф. В противном случае проводится поиск пересечений стороной треугольника препятствия. Если подобные пересечения найдены, в граф добавляется середина отрезка, не входящего в

препятствие. Затем добавляются грани, соответствующие всем связям добавленных в граф точек, не пересекающих препятствия. Вес грани определяется как евклидово расстояние.

```
    def create_graph(self, tri, points_array):
        G = nx.Graph()

        for simplex in tri.simplices:
            p1 = (points_array[simplex[0], 0], points_array[simplex[0], 1])
            p2 = (points_array[simplex[1], 0], points_array[simplex[1], 1])
            p3 = (points_array[simplex[2], 0], points_array[simplex[2], 1])

            centroid = ((p1[0] + p2[0] + p3[0]) / 3, (p1[1] + p2[1] + p3[1])
/ 3)
            v = []
            G.add_node(centroid)
            if not self.is_point_inside_obstacle(centroid):
                v.append(centroid)

            midpoints = [
                ((p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2),
                ((p2[0] + p3[0]) / 2, (p2[1] + p3[1]) / 2),
                ((p3[0] + p1[0]) / 2, (p3[1] + p1[1]) / 2)
            ]

            vertices = [p1, p2, p3]


            for i, (p_start, p_end) in enumerate([(p1, p2), (p2, p3), (p3,
p1)]):
                if not self.is_edge_on_obstacle(p_start, p_end):
                    midpoint = midpoints[i]
                    if not self.is_point_inside_obstacle(midpoint):
                        G.add_node(midpoint)
                        v.append(midpoint)
                    else:
                        # найти пересечение триангуляции с препятствием и
добавить точку середины отрезка
                        vert = self.find_all_line_vertices(p_start, p_end)
                        if vert:
                            for point in vert:
```

```
                                G.add_node(point)
                                v.append(point)

            # Добавляем вершины треугольника, если они не на препятствиях
            for point in vertices:
                if not self.is_point_on_obstacle(point) and not
self.is_point_inside_obstacle(point):
                    G.add_node(point)
                    v.append(point)

            for i in range(len(v)):
                for j in range(i + 1, len(v)):
                    if not self.find_intersection_with_obstacle(v[i], v[j]):
                        G.add_edge(v[i], v[j],
weight=np.linalg.norm(np.array(v[i]) - np.array(v[j])))

        G.add_node((self.start_point.x, self.start_point.y))
        G.add_node((self.finish_point.x, self.finish_point.y))

        return G
```

Поиск кратчайшего пути реализован при помощи метода, реализующего алгоритм Дейкстры. В случае, если путь из стартовой вершины в конечную не может быть найден, в консоль выводится соответствующее сообщение.

```
    def find_shortest_path(self, G):
        start_node = (self.start_point.x, self.start_point.y)
        finish_node = (self.finish_point.x, self.finish_point.y)
        try:
            path = nx.shortest_path(G, source=start_node, target=finish_node,
weight='weight')
            return path
        except nx.NetworkXNoPath:
            print(f"Нет пути между {start_node} и {finish_node}.")
            return None
```

5. Результаты работы

Для первого эксперимента количество вершин равно трём, размер 1, плотность препятствий является варьируемым параметром. На рисунке 12

изображен результат работы программы для плотности препятствий, равный 20.

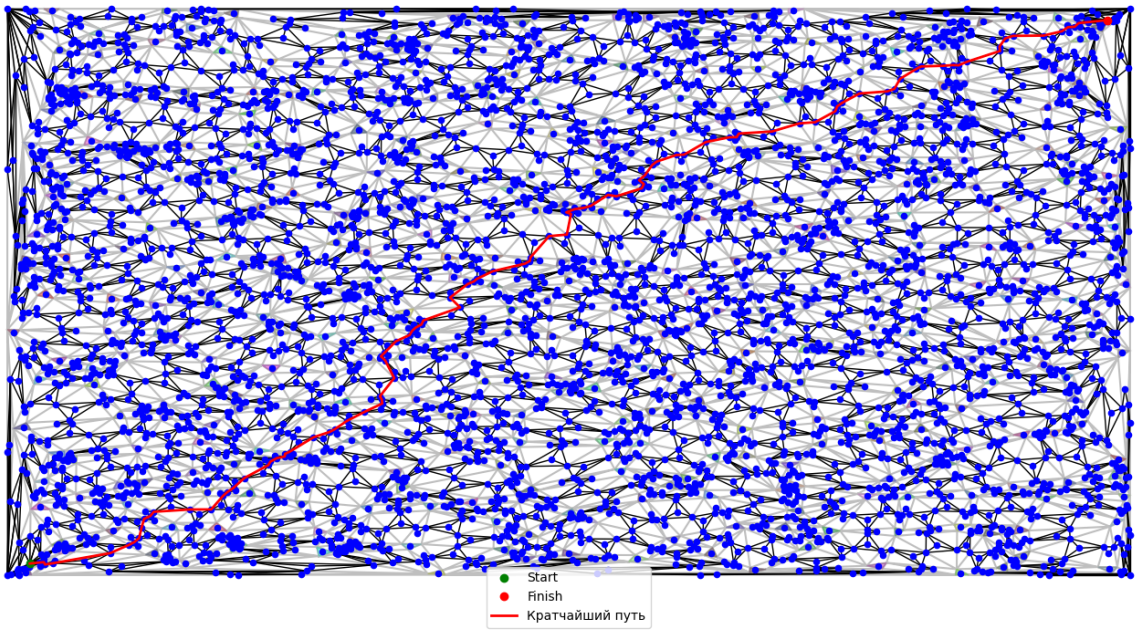


Рисунок 12 – Результат работы алгоритма с плотностью препятствий, равной 20.

Итоговое время выполнения для различных значений плотности препятствий:

Плотность препятствий	Время выполнения
3	0.025010108947753906
10	3.837907075881958
20	70.9803786277771
30	309.3849790096283

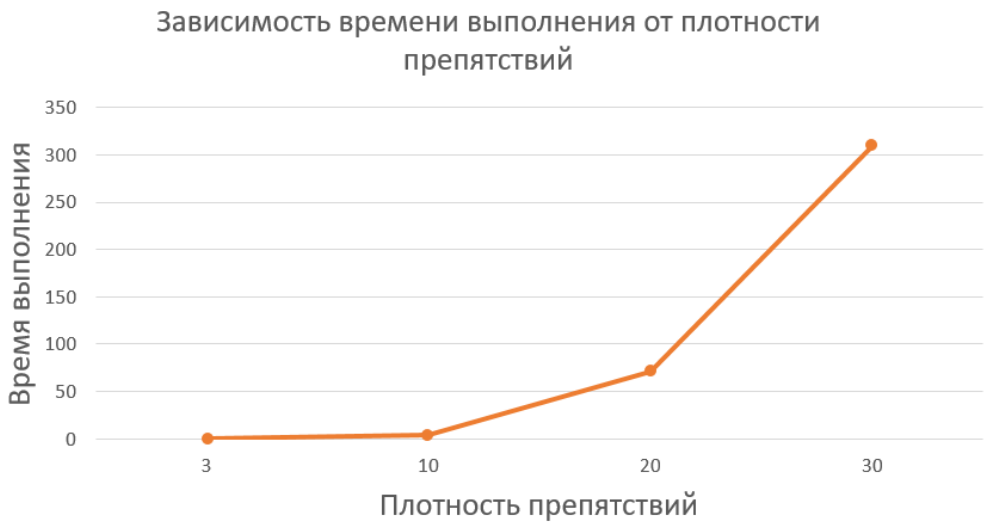


Рисунок 13 – Зависимость времени выполнения от плотности препятствий.

Из результатов следует, что сложность алгоритма является кубической. Кубическая сложность алгоритма наблюдается в алгоритме создания графа, где необходимо итерироваться по всем треугольникам триангуляции, затем итерироваться по граням треугольника с проверкой принадлежности к препятствию, после чего проверить принадлежность середины грани, в случае принадлежности произвести поиск середины отрезка, не принадлежащего препятствию.

Для второго эксперимента варьировалось количество вершин, плотность препятствий фиксирована значением 10, размер препятствий задан 5 для обеспечения наличия пути из начальной точки в целевую.

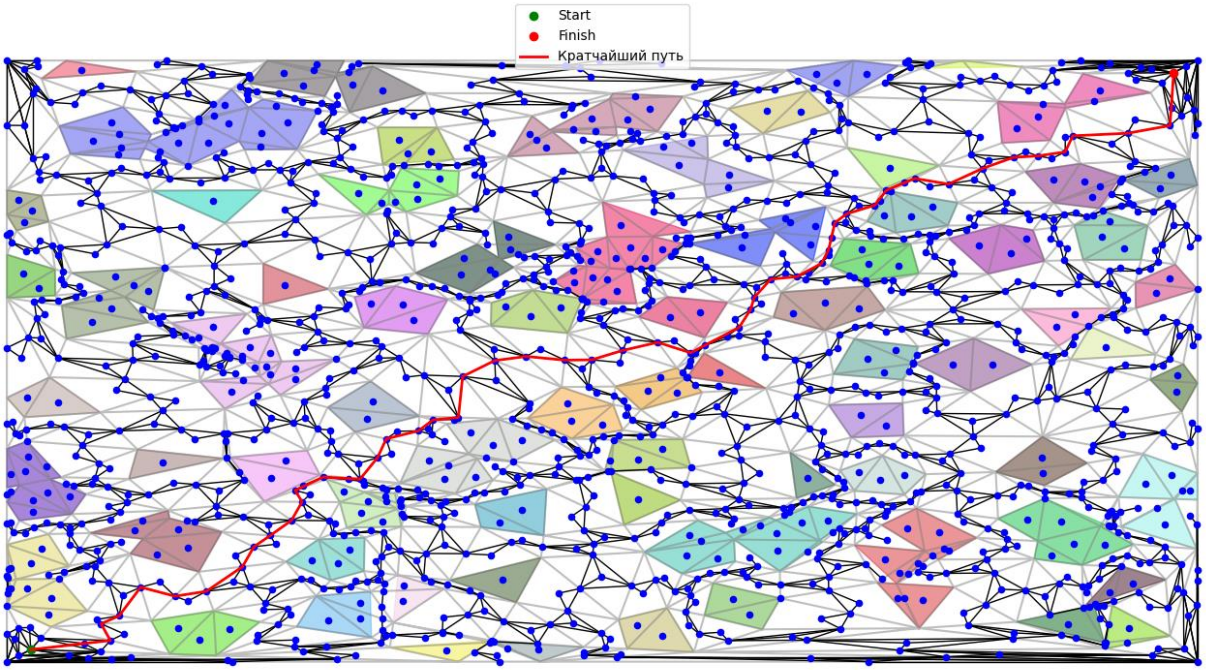


Рисунок 14 – Результат работы алгоритма с количеством вершин, равным 5.

Итоговое время выполнения для различных значений размера препятствий:

Количество вершин препятствий	Время выполнения
3	4.0912253856658936
5	5.209132194519043
7	7.496724843978882
10	9.901376962661743

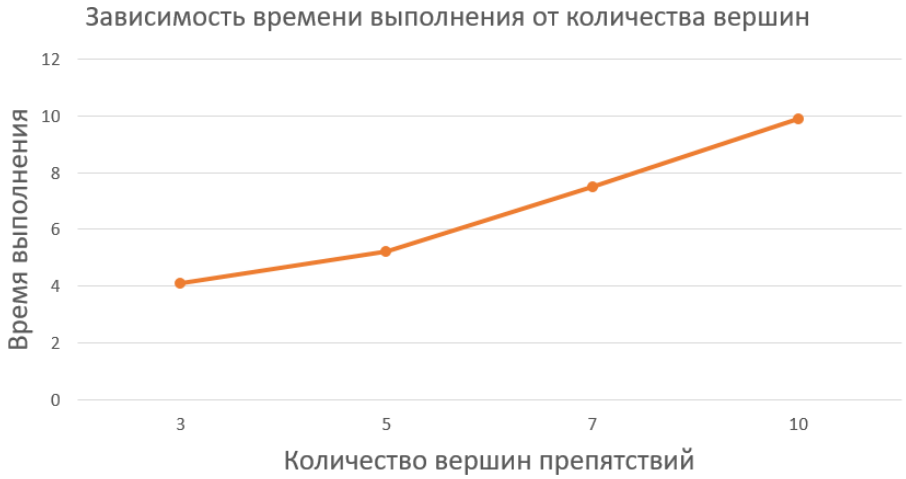


Рисунок 15 – Зависимость времени выполнения от количества вершин препятствий

Исходя из результатов, можно сделать вывод, что зависимость времени выполнения от количества вершин препятствий близка к линейной. Такой результат получается из-за того, что не все генерируемые препятствия содержат заданное число вершин (причина в работе генератора). В случае, если бы все препятствия создавались с заданным числом вершин, результат, представленный на рисунке 15, представлял бы собой зависимость, определённую сложностью алгоритма триангуляции. В данной реализации это сложность алгоритма Боуэра-Ватсона, то есть $O(N^2)$.

В завершающем эксперименте проводилось варьирование размера препятствий. Поскольку количество вершин и плотность препятствий остаётся постоянным, повышение времени решения может возникать только из-за повышения количества треугольников триангуляции, не заполненных препятствием полностью. Поэтому этот эксперимент проводиться больше для проверки улучшения алгоритма составления графа, ведь в классической реализации (при выделении только характерных точек) есть вероятность получения несвязанного графа.

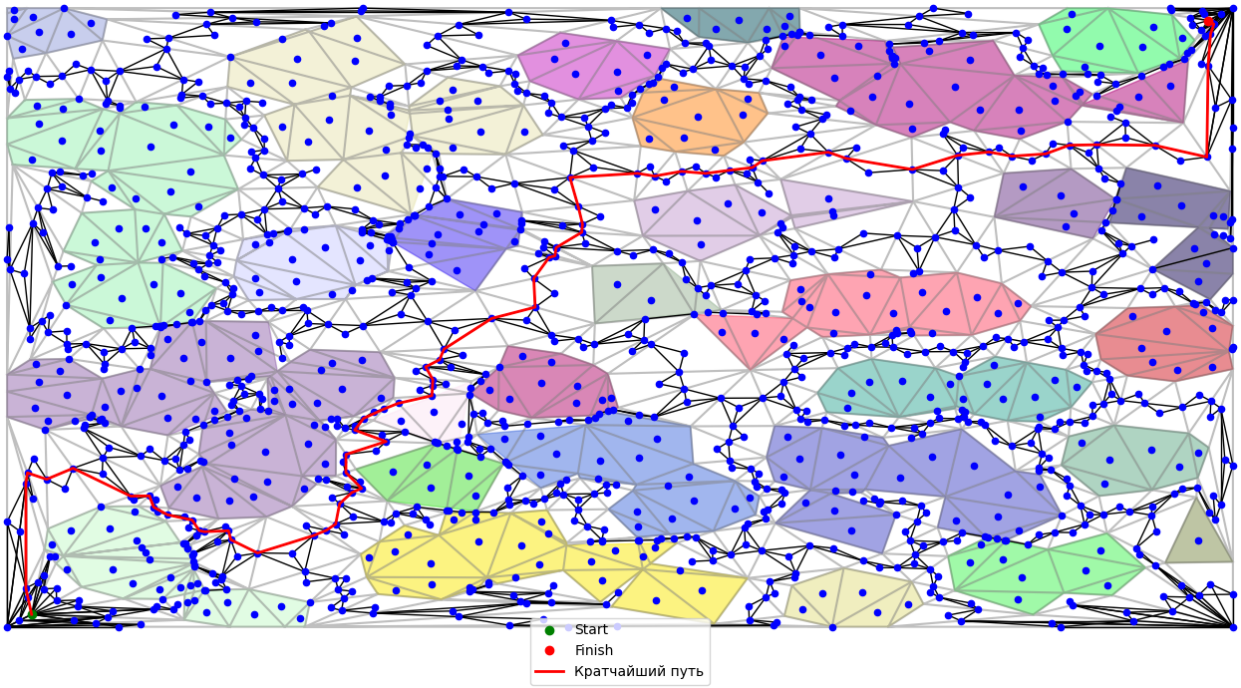


Рисунок 16 – Результат работы алгоритма при плотности препятствий 8 и размере 7.

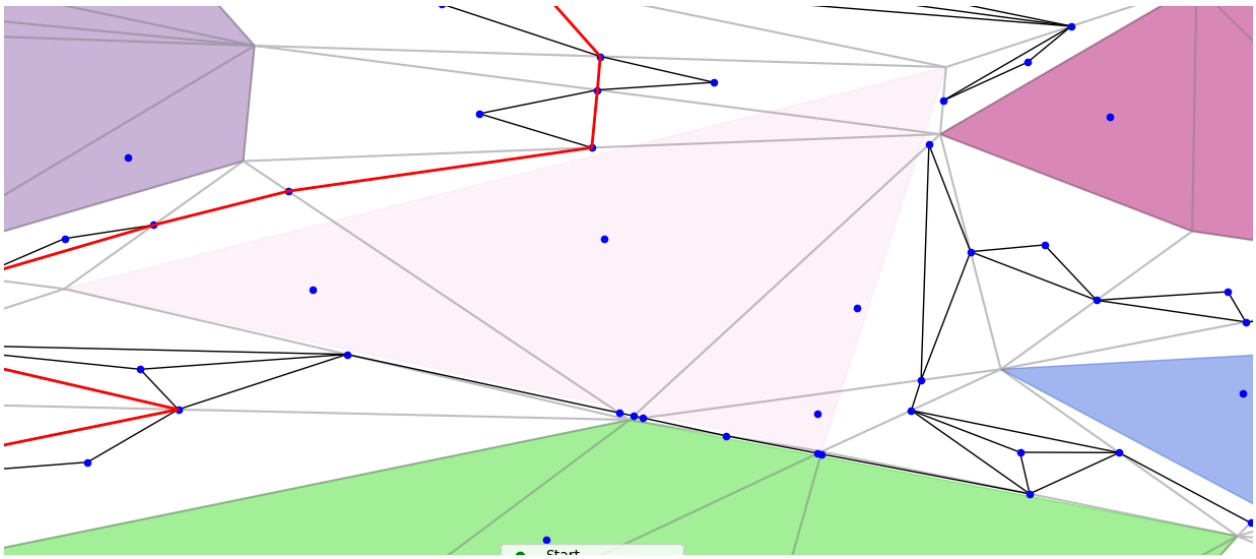


Рисунок 17 – Пример треугольников триангуляции, частично заполненных препятствием.

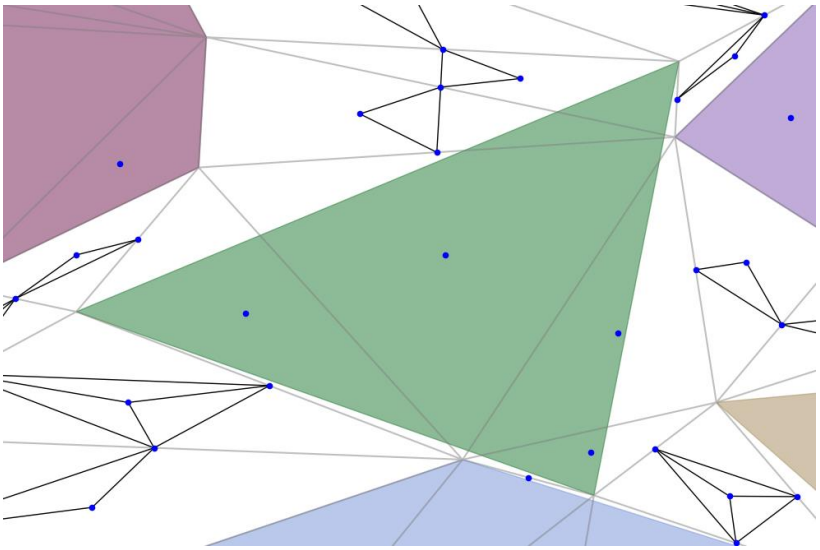
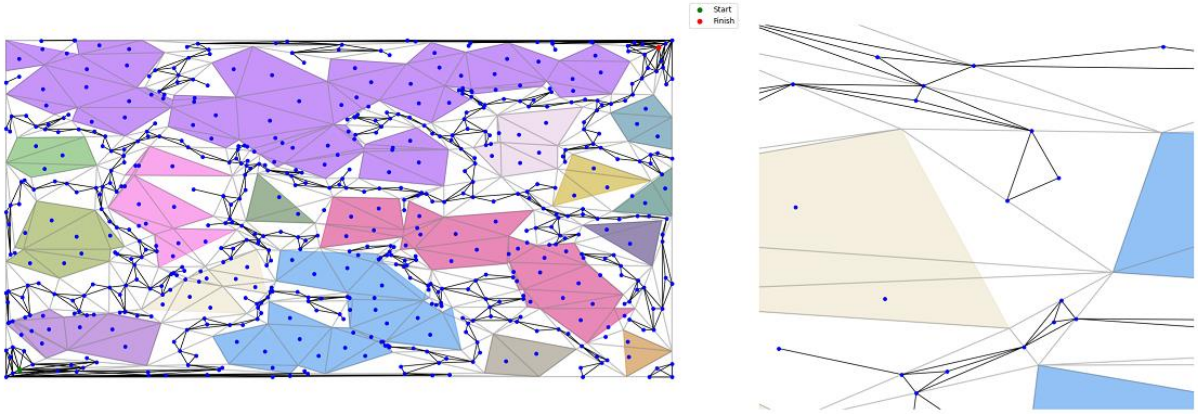


Рисунок 18 – Пример классической реализации построения графа.

На рисунках 17-18 приведено отличие классической реализации определения графа и улучшенной. В отдельных случаях отсутствие нехарактерных точек триангуляции приведёт к несвязности графа, из-за чего путь между начальной и целевой вершинами не будет найден.

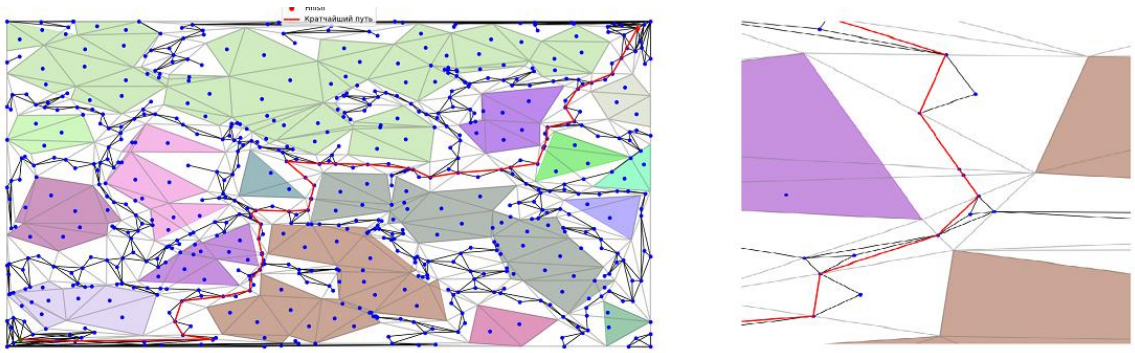
Пример случая получения несвязного графа представлен на рисунке 19.



а) б)

Рисунок 19 – Пример получения несвязного графа: а) результат работы алгоритма, б) место разрыва пути между начальной точкой и целевой.

Улучшенный же алгоритм, в свою очередь нашёл кратчайший путь. Результат работы на том же пространстве представлен на рисунке 20.



а) б)

Рисунок 20 – а) результат работы алгоритма с улучшенным алгоритмом создания графа, б) место разрыва пути между начальной точкой и целевой при использовании классической реализации.

Заключение

В результате выполненной работы были изучены и реализованы алгоритмы триангуляции пространства Боуэра-Ватсона, слияния препятствий Уайлера-Атертона, составления графа.

В результате проведённых экспериментов были получены оценки времени выполнения от различных параметров препятствий конфигурационного пространства. Была доказана и продемонстрирована необходимость усовершенствования алгоритма составления графа. Результирующая сложность работы алгоритма оценивается как кубическая.

Список литературы

1. Скворцов А. В. Обзор алгоритмов построения триангуляции Делоне // Вычислительные методы и программирование. 2002. Т.3.
2. Manacher G., Zobrist A. Neither the Greedy nor the Delaunay triangulation of planar point set approximates the optimal triangulation // Inf. Proc. Let. 1977. 9, N 1. 31-43.
3. Bowyer A Computing Dirichlet Tessellations The Computer Journal Vol. 24, No. 2, 1981.
4. WatsonD Computing the ndimensional Delaunay tessellation with appli cations to Vornoi polytopes The Computer Journal Vol. 24, No. 2, 1981.
5. Kevin Weiler and Peter Atherton. 1977. Hidden surface removal using polygon area sorting. SIGGRAPH Comput. Graph. 11, 2 (Summer 1977), 214–222. https://doi.org/10.1145/965141.563896