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, осуществляющий триангуляцию свободного пространства и поиск кратчайшего пути в заданном двумерном пространстве препятствий.

2. Алгоритм

Триангуляцией 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 $O(e^N)$. 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.

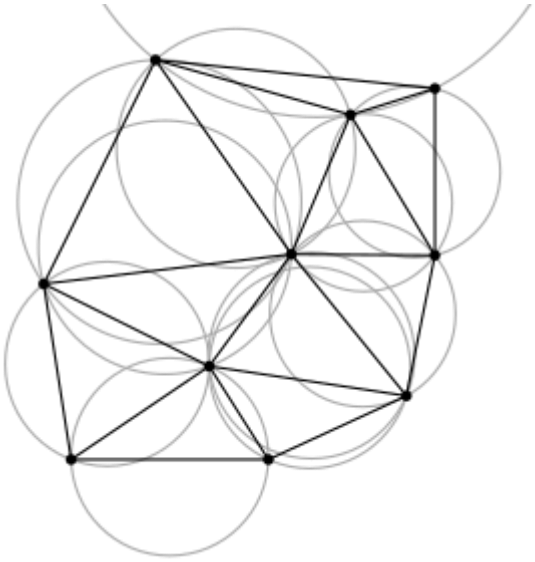

Рисунок 1 – пример триангуляции Делоне

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 $O(N^2)$.

Алгоритм Дейкстры начинается с установки начальной вершины и работы от этой точки. Он работает по принципу «жадного» алгоритма, что означает, что на каждом шаге он стремится минимизировать текущую общую стоимость пути.

Сначала инициализируются два множества:

- Множество, содержащее уже обработанные вершины (изначально пустое).
- Множество, содержащее все остальные вершины графа (изначально содержит все вершины графа).

Также каждой вершине графа присваивается вес, который представляет минимальную известную стоимость пути от начальной вершины до данной. Для начальной вершины этот вес равен 0, для всех остальных вершин — бесконечность.

На каждом шаге алгоритм выбирает вершину из непосещенного множества с наименьшим весом, перемещает эту вершину в множество посещенных вершин и обновляет веса всех соседей выбранной вершины. Вес соседа обновляется, если через выбранную вершину можно добраться до этого соседа с меньшей стоимостью.

Процесс продолжается, пока не будут посещены все вершины или пока мы не найдем путь до конечной вершины (если она задана).

3. Реализация

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 и препятствия.

```
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

    def print(self) -> None:
        print(self.x, self.y)

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

Класс точки реализует задание как с помощью координат, так и при помощи словаря. В препятствии случайно генерируется цвет для последующего отображения.

Основным для алгоритма является реализованный класс поля с описанными методами триангуляции свободного пространства, построения графа и нахождения кратчайшего пути.

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


    def triangulate_free_space(self):
        all_points = []
        for obstacle in self.obstacles:
            all_points.extend([(point.x, point.y) for point in
obstacle.points])


        field_boundary = [(0, 0), (100, 0), (100, 100), (0, 100)]
        all_points.extend(field_boundary)
        all_points.append((self.start_point.x, self.start_point.y))
        all_points.append((self.finish_point.x, self.finish_point.y))
        points_array = np.array(all_points)

        tri = Delaunay(points_array)
        if PLOT:
            fig, ax = plt.subplots()
            ax.triplot(points_array[:, 0], points_array[:, 1], tri.simplices)
            ax.plot(points_array[:, 0], points_array[:, 1], 'o')

            for obstacle in self.obstacles:
                x_coords, y_coords = zip(*[(point.x, point.y) for point in
obstacle.points])
                ax.fill(x_coords, y_coords, color=obstacle.color, alpha=0.5)

            ax.set_xlabel('X')
            ax.set_ylabel('Y')
            ax.grid(True)
            plt.show()

        return tri, points_array

    def add_point(self, point: Point) -> None:
        self.points.append(point)
```

```
    def print(self) -> None:
        for point in self.points:
            point.print()

    def draw_obstacles(self):
        fig, ax = plt.subplots()

        ax.plot(self.start_point.x, self.start_point.y, 'go', label='Start')
        ax.plot(self.finish_point.x, self.finish_point.y, 'ro',
label='Finish')

        for obstacle in self.obstacles:
            x_coords, y_coords = zip(*[(point.x, point.y) for point in
obstacle.points])
            ax.fill(x_coords, y_coords, color=obstacle.color, alpha=0.5)

        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.legend()
        ax.grid(True)
        plt.show()

    def is_edge_on_obstacle(self, p1, p2):
        for obstacle in self.obstacles:
            obstacle_edges = [(obstacle.points[i].x, obstacle.points[i].y,
obstacle.points[(i+1)%len(obstacle.points)].x,
obstacle.points[(i+1)%len(obstacle.points)].y) for i in
range(len(obstacle.points))]
            for edge in obstacle_edges:
                if (p1[0] == edge[0] and p1[1] == edge[1] and p2[0] ==
edge[2] and p2[1] == edge[3]) or (p1[0] == edge[2] and p1[1] == edge[3] and
p2[0] == edge[0] and p2[1] == edge[1]):
                    return True
        return False

    def is_point_on_obstacle(self, point):
        for obstacle in self.obstacles:
            if point in [(p.x, p.y) for p in obstacle.points]:
                return True
        return False

    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)
            G.add_node(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]
            v = []

            if not self.is_edge_on_obstacle(p1, p2):
                G.add_node(midpoints[0])
                G.add_edge(centroid, midpoints[0],
weight=np.linalg.norm(np.array(centroid) - np.array(midpoints[0])))
                v.append(midpoints[0])

            if not self.is_edge_on_obstacle(p2, p3):
                G.add_node(midpoints[1])
                G.add_edge(centroid, midpoints[1],
weight=np.linalg.norm(np.array(centroid) - np.array(midpoints[1])))
                v.append(midpoints[1])

            if not self.is_edge_on_obstacle(p1, p3):
                G.add_node(midpoints[2])
                G.add_edge(centroid, midpoints[2],
weight=np.linalg.norm(np.array(centroid) - np.array(midpoints[2])))
                v.append(midpoints[2])

            for point in vertices:
                if not self.is_point_on_obstacle(point):
                    G.add_node(point)
```

```
                    G.add_edge(centroid, point,
weight=np.linalg.norm(np.array(centroid) - np.array(point)))
                    v.append(point)


            for i in range(len(v)):
                for j in range(i + 1, len(v)):
                    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


    def draw_graph(self, G, tri, points_array):
        pos = {node: node for node in G.nodes()}
        fig, ax = plt.subplots()


        for obstacle in self.obstacles:
            x_coords, y_coords = zip(*[(point.x, point.y) for point in
obstacle.points])
            ax.fill(x_coords, y_coords, color=obstacle.color, alpha=0.5)


        ax.triplot(points_array[:, 0], points_array[:, 1], tri.simplices,
color='gray', alpha=0.5)


        nx.draw(G, pos, with_labels=False, node_size=20, node_color='blue',
ax=ax)
        ax.plot(self.start_point.x, self.start_point.y, 'go', label='Start')
        ax.plot(self.finish_point.x, self.finish_point.y, 'ro',
label='Finish')
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.legend()
        ax.grid(True)
        plt.show()


    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)
        path = nx.shortest_path(G, source=start_node, target=finish_node,
weight='weight')
```

```
        return path
```

Для запуска алгоритма и получения кратчайшего пути необходимо определить начальную и целевую точки, а также препятствия.

```
start = Point(dict=start_points)
finish = Point(dict=end_points)
edges = [[0, 0], [100, 0], [0, 100], [100, 100]]
edges_points = list(map(lambda coords: Point(x=coords[0], y=coords[1]), edges))
obstacles = []

for polygon in polygons:
    points = polygon.get('points')
    points = list(map(lambda point: Point(x=point.get('x'), y=point.get('y')), points))
    obstacle = Obstacle(points)
    obstacles.append(obstacle)

field = Field(start, finish, edges_points, obstacles)

if PLOT:
    field.draw_obstacles()
tri, points_array = field.triangulate_free_space()

G = field.create_graph(tri, points_array)
if PLOT:
    field.draw_graph(G, tri, points_array)

shortest_path = field.find_shortest_path(G)
print("Кратчайший путь:", shortest_path)
```

4. Результаты работы

В результате работы алгоритма были получены графики: исходного пространства, триангуляции свободного пространства, итогового графа с кратчайшим путём для препятствий разной плотности.

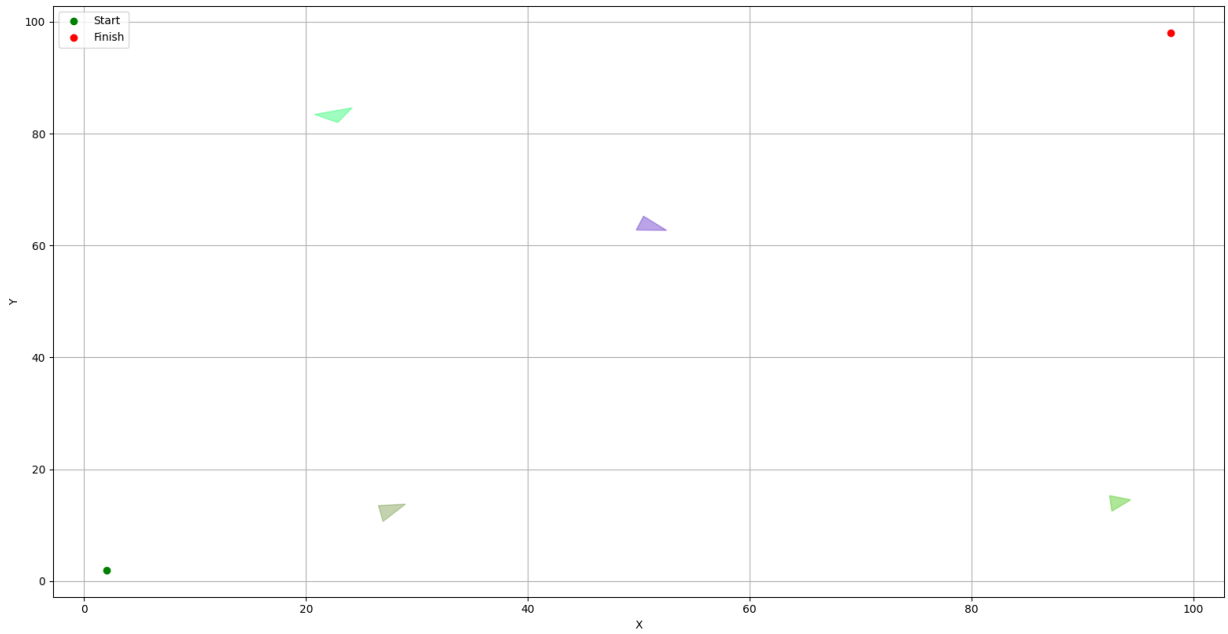

Рисунок 1 - Исходное пространство.

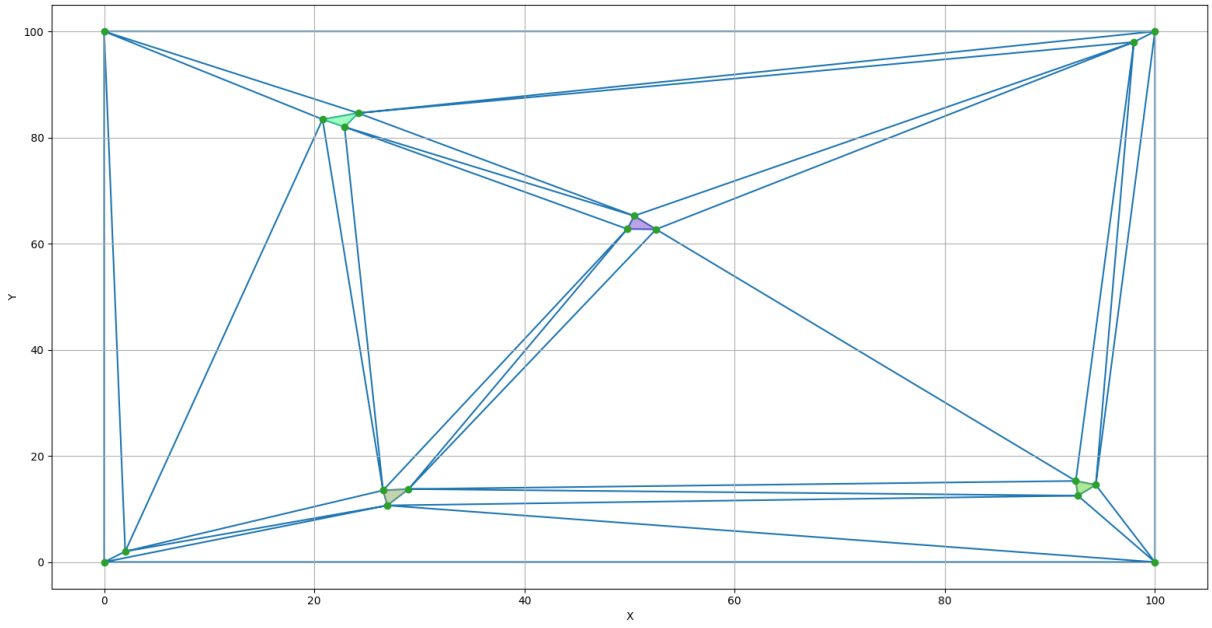

Рисунок 2 – Триангуляция свободного пространства.

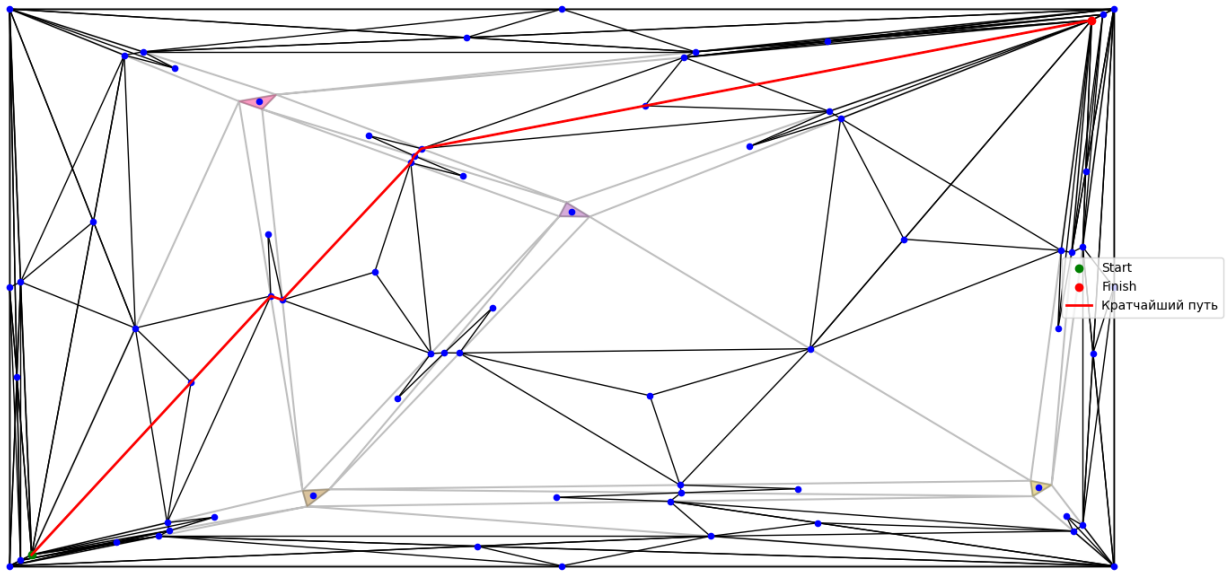

Рисунок 3 – Итоговый граф и кратчайший путь.

Для пространства со средней плотностью препятствий:

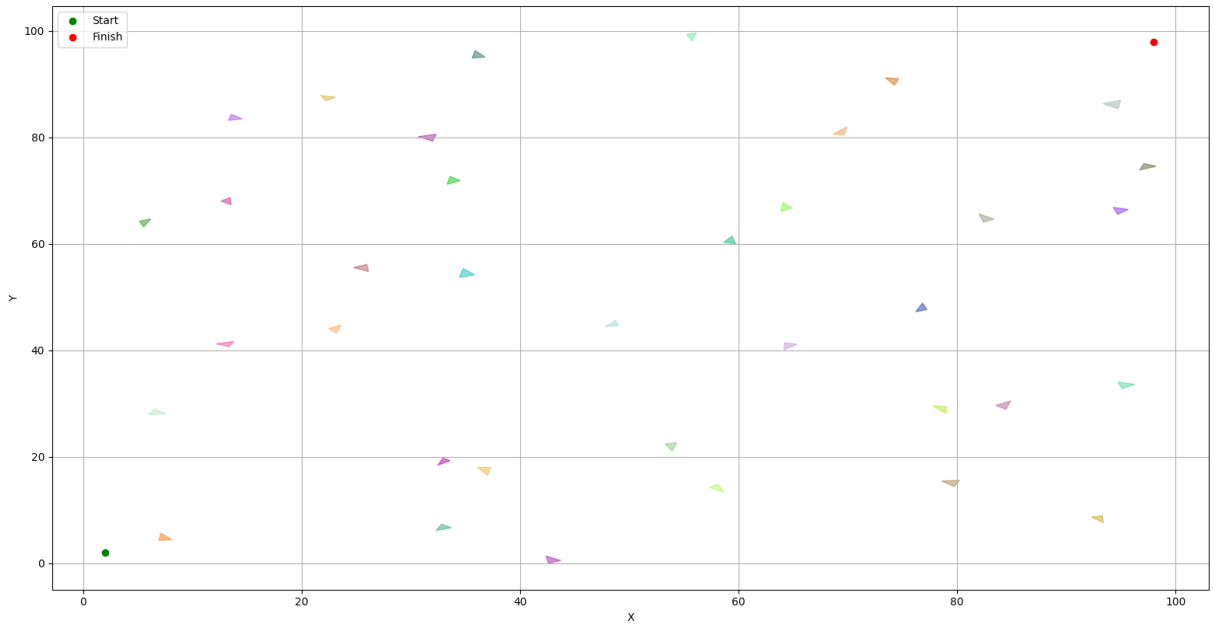

Рисунок 4 – Исходное пространство.

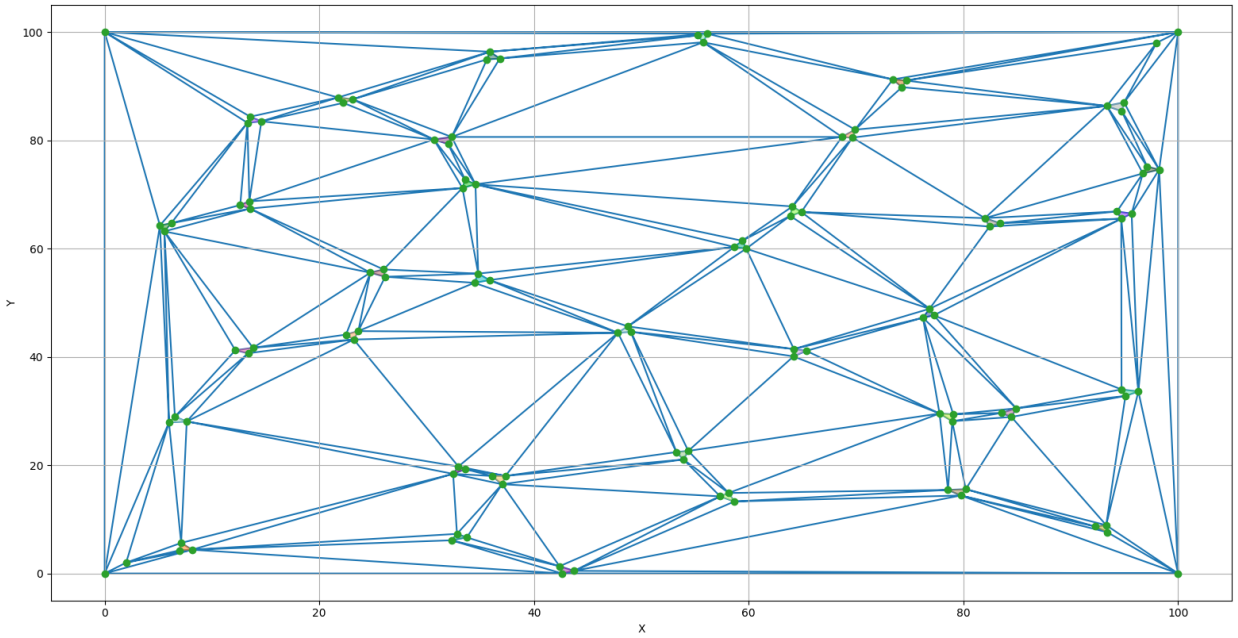

Рисунок 5 – Триангуляция свободного пространства

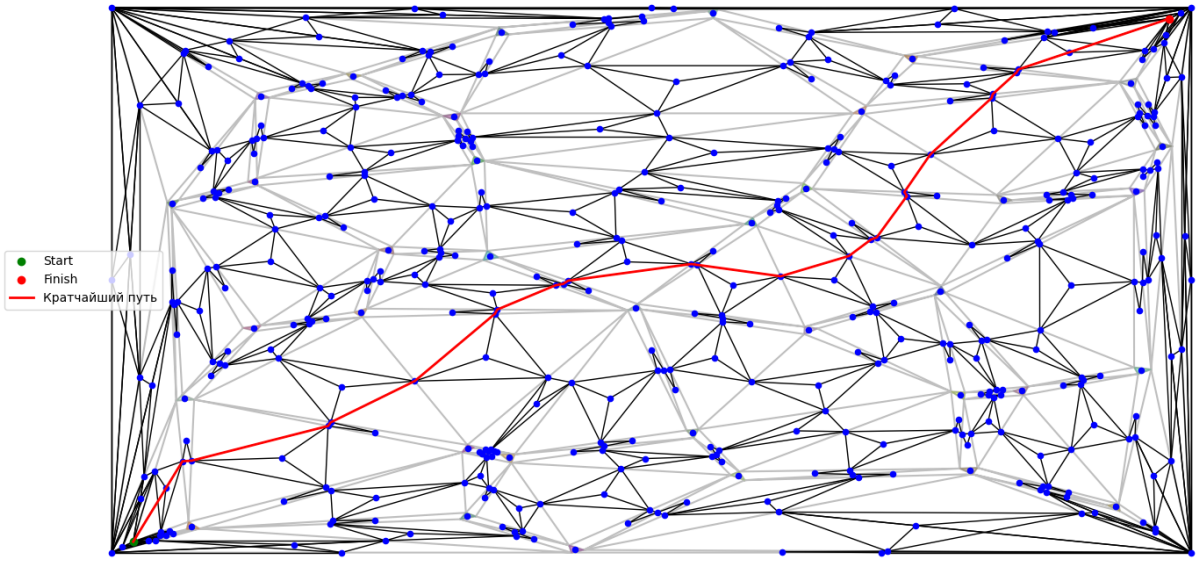

Рисунок 6 – Итоговый граф и кратчайший путь.

Для пространства с высокой плотностью препятствий:

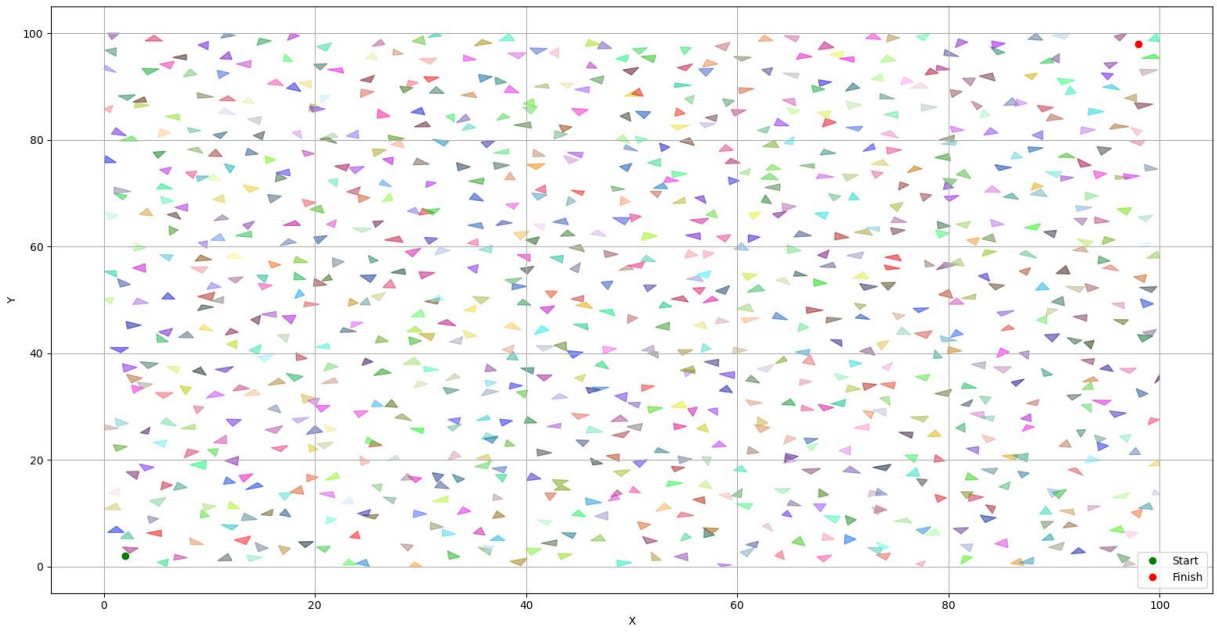

Рисунок 7 – Исходное пространство.

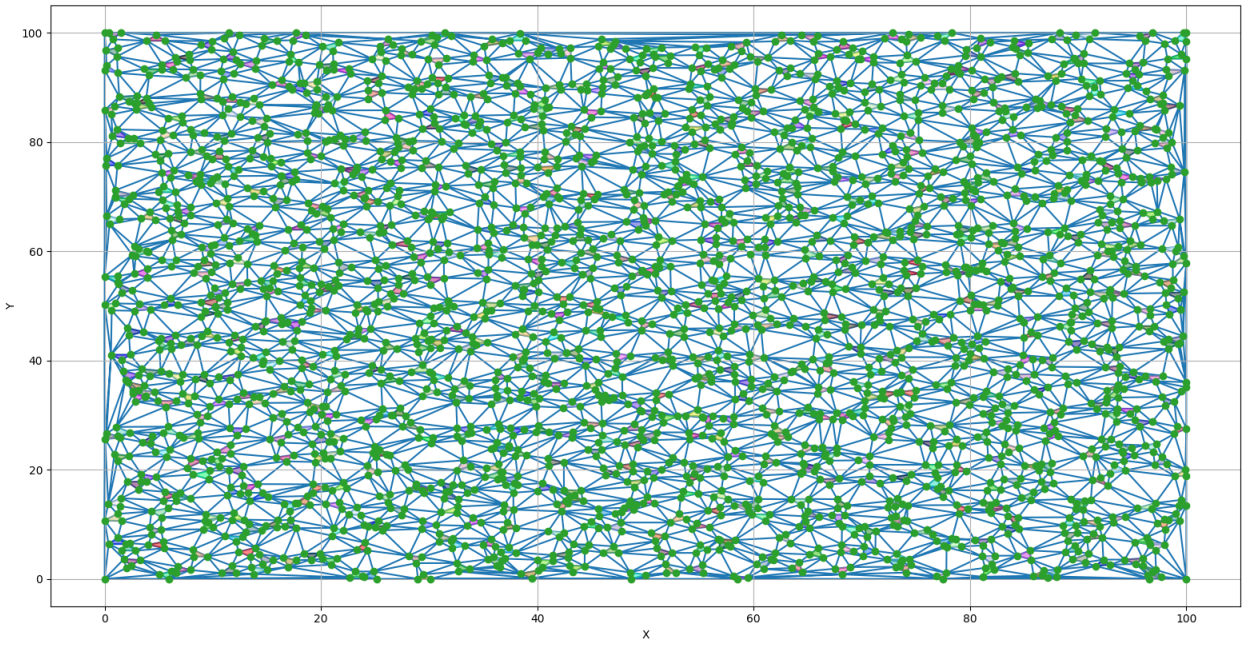

Рисунок 8 – Триангуляция свободного пространства.

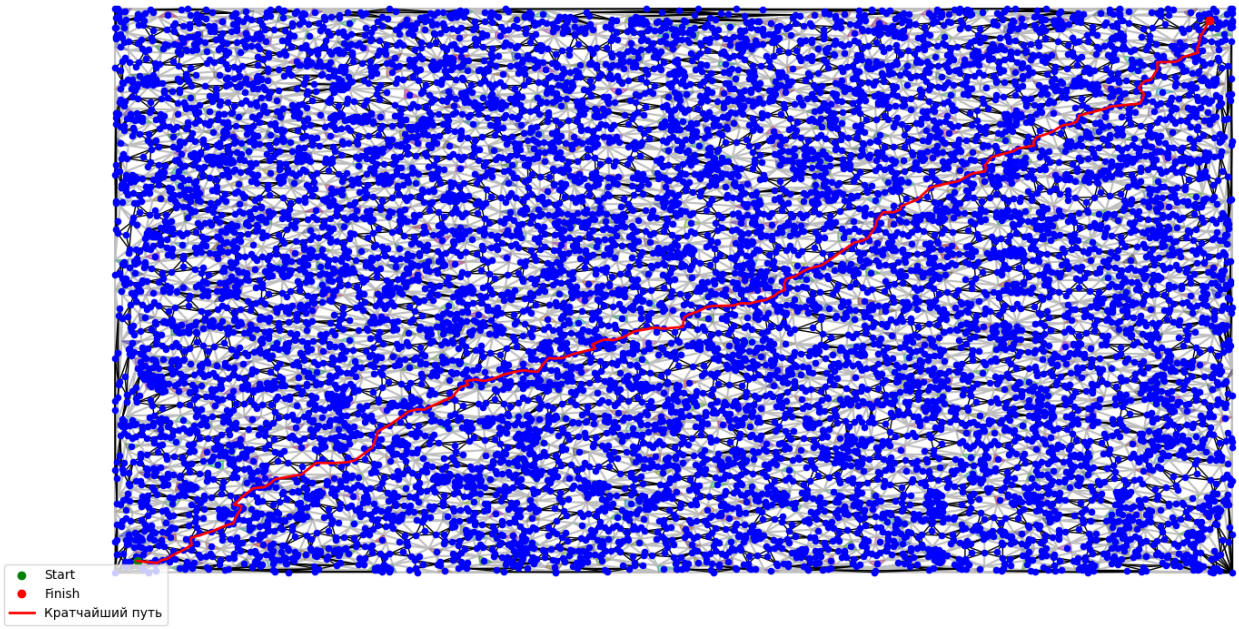

Рисунок 9 – Итоговый граф и кратчайший путь

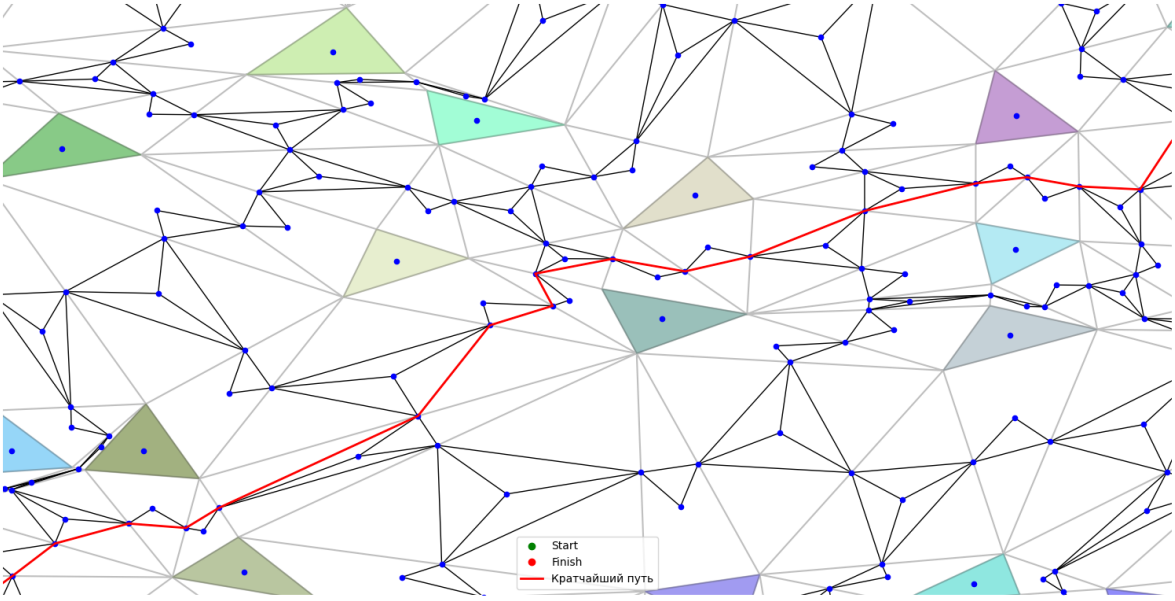

Рисунок 10 – Итоговый граф и кратчайший путь

Итоговое время выполнения:

Плотность препятствий	Время выполнения
3	0.0049970149993896484
10	0.4054696559906006
30	28.23429536819458
40	96.65564632415771

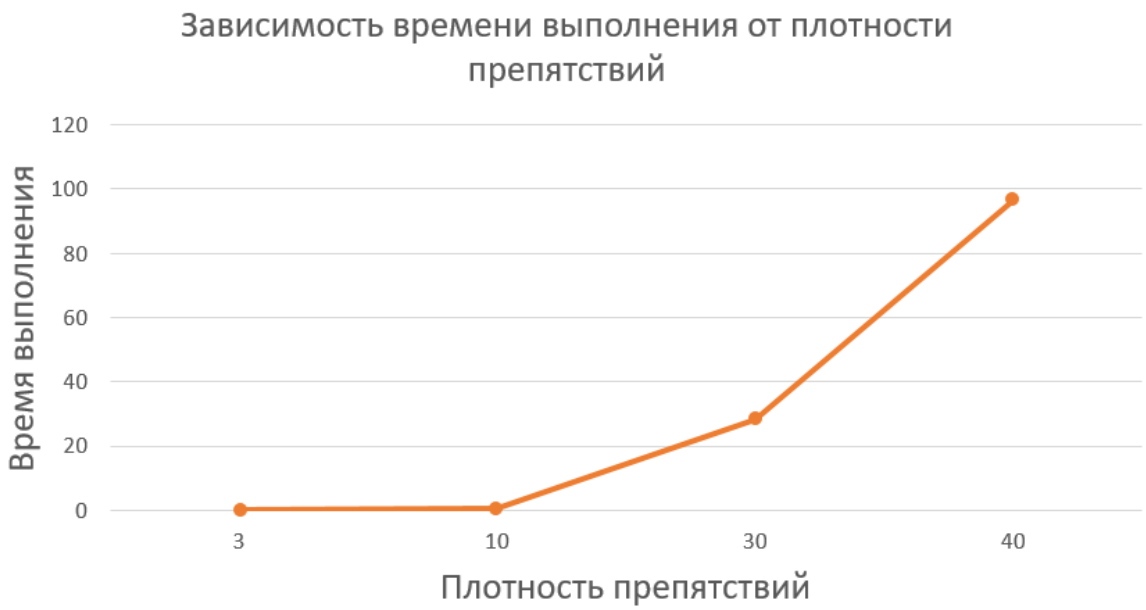

Рисунок 11 – Зависимость времени выполнения от плотности препятствий.