**Министерство науки и высшего образования РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Уфимский университет науки и технологий»**

**Кафедра** Высокопроизводительных вычислений и дифференциальных уравнений

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 100 |  |  |  |  |  |  |  |  |  |  |
| 90 |  |  |  |  |  |  |  |  |  |  |
| 80 |  |  |  |  |  |  |  |  |  |  |
| 70 |  |  |  |  |  |  |  |  |  |  |
| 60 |  |  |  |  |  |  |  |  |  |  |
| 50 |  |  |  |  |  |  |  |  |  |  |
| 40 |  |  |  |  |  |  |  |  |  |  |
| 30 |  |  |  |  |  |  |  |  |  |  |
| 20 |  |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  |
| 0 |  |  |  |  |  |  |  |  |  |  |

**СРАВНИТЕЛЬНЫЙ АНАЛИЗ АЛГОРИТМОВ ДЛЯ ПЛАНИРОВАНИЯ МАРШРУТОВ ТРАНСПОРТНЫХ СРЕДСТВ С УЧЕТОМ РЕЛЬЕФА МЕСТНОСТИ**

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

«Численные методы»

**2342.233221.000 ПЗ**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа  МКН-318Б | Фамилия И.О. | Подпись | Дата | Оценка |
| Студент | Нечаев Б.П. |  |  |  |
| Консультант | Касаткин А.А. |  |  |  |
| Принял | Лукащук В.О. |  |  |  |

Уфа 2024

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Уфимский университет науки и технологий»

Кафедра Высокопроизводительных вычислений и дифференциальных уравнений

**ЗАДАНИЕ**

на курсовой проект по дисциплине

**«Численные методы»**

Студент: Нечаев Борис Павлович Группа: МКН-318Б

Консультант: Касаткин Алексей Александрович

1. Тема курсового проекта

Сравнительный анализ алгоритмов для планирования маршрутов транспортных средств с учетом рельефа местности.

2. Основное содержание

2.1. Изучить алгоритмы для планирования маршрутов.

2.2. Реализовать алгоритмы поиска маршрутов с учетом рельефа местности.

2.3. Произвести сравнение реализованных алгоритмов.

2.4. Оформить пояснительную записку к курсовому проекту.

3. Требования к оформлению материалов работы

Требования к оформлению пояснительной записки

Пояснительная записка к курсовому проекту должна быть оформлена в соответствии с требованиями ГОСТ и содержать

• титульный лист,

• задание на курсовой проект,

• содержание,

• введение,

• теоретическую часть, описывающую алгоритм

• практическую часть, описывающую особенности программной реализации и результаты исследования

• заключение,

• список литературы,

• приложение, содержащее листинг разработанной программы.

|  |  |
| --- | --- |
| Дата выдачи задания  "\_\_" \_\_\_\_\_\_\_\_ 202\_ г. | Дата окончания работы  "\_\_" \_\_\_\_\_\_\_\_\_\_ 202\_ г. |

Консультант \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Касаткин А.А.

**Содержание**

[Введение 5](#_Toc185263664)

[1. Теоретическая часть 6](#_Toc185263665)

[1.1. Математическая постановка задачи 6](#_Toc185263666)

[1.2. Алгоритм А\*(A-star) 7](#_Toc185263667)

[1.3. Алгоритм ближайшего соседа 9](#_Toc185263668)

[1.4. Алгоритм имитации отжига 9](#_Toc185263669)

[1.5. Генетический алгоритм 12](#_Toc185263670)

[2. Практическая часть 16](#_Toc185263671)

[2.1. Общий вид программы 16](#_Toc185263672)

[2.2. Алгоритм А\* 17](#_Toc185263673)

[2.3. Алгоритм грубой силы 20](#_Toc185263674)

[2.4. Алгоритм ближайшего соседа 21](#_Toc185263675)

[2.5. Алгоритм имитации отжига 23](#_Toc185263676)

[2.6. Генетический алгоритм 26](#_Toc185263677)

[2.7. Полученные результаты 29](#_Toc185263678)

[Заключение 30](#_Toc185263679)

[Список литературы 31](#_Toc185263680)

[Приложение А 32](#_Toc185263681)

**Введение**

В современном обществе, где логистика и управление транспортом имеют большое значение для бизнеса и экономики, задачи маршрутизации транспортных средств становятся все более важными. Особенно актуальными являются задачи, в которых необходимо учитывать не только оптимальность маршрута, но и особенности рельефа местности. Эффективное решение этих задач может значительно сократить затраты на топливо, время в пути и повысить общую эффективность транспортных процессов. Одной из подобных задач является задача коммивояжера (TSP), в которой необходимо найти кратчайший маршрут, проходящий через все заданные точки. Эта задача является NP-трудной, что означает отсутствие эффективных алгоритмов для ее решения в общем случае. Однако для ее решения применяются различные эвристические методы, такие как генетические алгоритмы, алгоритм имитации отжига и другие.

**Цель работы:** проведение сравнительного анализа различных алгоритмов для планирования маршрутов транспортных средств с учетом рельефа местности, а также их реализация и оценка эффективности в контексте задачи маршрутизации транспортных средств (VRP).

**Задачи работы:**

* Изучить и сравнить алгоритмы для планирования маршрутов с учетом рельефа местности.
* Реализовать алгоритмы маршрутизации с учетом различных факторов, включая перепады высот.
* Произвести сравнение эффективности реализованных алгоритмов, оценив их быстродействие и точность.

1. **Теоретическая часть**

## Математическая постановка задачи

Задача маршрутизации транспортных средств (VRP) представляет собой задачу комбинаторной оптимизации и целочисленного программирования, которая решает вопрос: «Какова оптимальная совокупность маршрутов для автопарка, которые должны быть пройдены для доставки товара определённым клиентам?»

Эта задача является обобщением задачи коммивояжёра (TSP) и впервые была представлена в статье Джорджа Данцига и Джона Рамсера в 1959 году, где был предложен первый алгоритмический подход, применённый к доставке топлива [1]. Обычно контекст задачи связан с доставкой товаров от одного или нескольких складов (депо) к клиентам, которые разместили заказы. Цель VRP — минимизация общей стоимости маршрутов, которая может измеряться в денежном выражении, расстоянии или времени.

Маршрут в контексте VRP определяется порядком посещения точек. При этом перемещение между точками двумя осуществляется по кратчайшему пути, который рассчитывается с учётом рельефа местности и других факторов. В данной работе для вычисления стоимости пути между точками используется алгоритм A\*.

**Математическая формулировка задачи**:

**Множество точек** где — начальная точка, — конечная точка, ​ — промежуточные точки маршрута [4].

**Граф местности** может быть представлен как где:

* V — множество вершин, представляющих точки на местности (в том числе начальную и конечную точки).
* E — множество рёбер, которые соединяют эти точки. Каждое ребро имеет ассоциированную стоимость пути.

**Стоимость ребра** ), где функция определяет стоимость пути между точками и ​, учитывая различные параметры, такие как рельеф местности.

**Целевая функция:**

Минимизировать общую стоимость маршрута:

где T— множество рёбер маршрута, а — стоимость каждого ребра.

**Ограничения**:

* Транспортное средство должно начать и завершить маршрут в заранее заданных точках и
* Каждая промежуточная точка должна быть посещена ровно один раз.

**Алгоритмические подходы**

Для решения задачи VRP могут применяться различные методы:

* **Точные методы:** Например, метод перебора и целочисленного программирования, которые обеспечивают точное решение, но могут быть неэффективны для больших задач.
* **Эвристики:** Такие как жадные алгоритмы, генетические алгоритмы, симулированный отжиг и алгоритмы на основе роя частиц. Эти методы могут дать хорошее решение за разумное время, но не гарантируют нахождение глобального оптимума.
* **Методы на основе графов:** Использование графовых структур для поиска оптимальных маршрутов, таких как алгоритмы поиска кратчайших путей или преобразование графов.

Задача VRP представляет собой классическую задачу оптимизации, с многими вариациями в зависимости от конкретных условий и ограничений, таких как максимизация прибыли, минимизация времени или расстояния, а также учёт особенностей местности и транспортных средств.

В данной работе будут реализованы следующие методы маршрутизации: алгоритм ближайшего соседа, генетический алгоритм, симулированный отжиг. Все эти методы будут использовать алгоритм A\* для нахождения стоимости ребра маршрута.

## Алгоритм А\*(A-star)

В данном проекте алгоритм A\* применяется для поиска оптимального пути между двумя точками на карте.

Алгоритм A\* — это метод поиска кратчайшего пути, который находит оптимальный маршрут между начальной и конечной точкой в графе. В отличие от алгоритма Дейкстры, который анализирует все возможные пути, A\* использует эвристическую функцию для ускорения поиска, что позволяет находить решение быстрее [5].

Расстояние между точками в A\* обычно определяется с использованием евклидова расстояния (или Манхэттенского расстояния, в зависимости от типа задачи) [1]. Однако, в контексте данной работы, к этому расстоянию добавляется слагаемое, которое учитывает перепады высот на пути.

Ключевым аспектом работы алгоритма A\* является выбор подходящей эвристической функции. Эвристика должна быть "достаточно хорошей", чтобы алгоритм был эффективным, но при этом гарантировалась оптимальность найденного пути. Наиболее распространенные эвристики:

* Манхэттенское расстояние (для движения по сетке):
* Евклидово расстояние:

Алгоритм A\* использует три ключевых функции для оценки стоимости пути:

* g(n) — стоимость пути от начальной вершины до текущей вершины n.
* h(n) — эвристическая функция, которая оценивает оставшуюся стоимость пути от текущей вершины до целевой.
* f(n) = g(n) + h(n) — общая оценка стоимости пути через вершину n. Это значение используется для сортировки узлов по приоритету.

Алгоритм A\* начинает с начальной вершины и повторно выбирает вершины для исследования на основе минимальной суммы значений f(n). Он использует два списка:

* Открытый список (open): содержит вершины, которые должны быть исследованы.
* Закрытый список (closed): хранит уже исследованные вершины.

Алгоритм работает следующим образом:

**A\*(start, goal)**

**1 closed ← пустое множество**

**2 open ← создать\_очередь(f)**

**3 добавить(open, путь(start))**

**4 пок open не пуста**

**5 p ← извлечь\_первый(open)**

**6 x ← последний узел пути p**

**7 если x in closed**

**8 continue**

**9 если x == goal**

**10 return p**

**11 добавить(closed, x)**

**12 для каждого сосед y вершины x**

**13 если y не в closed**

**14 добавить(open, добавить\_к\_пути(p, y))**

**15 вернуть неудача**

1. Начинается с начальной вершины, которая добавляется в открытый список.
2. Из открытого списка выбирается вершина с минимальной оценкой f(n).
3. Эту вершину перемещают в закрытый список, а ее соседей добавляют в открытый список.
4. Процесс продолжается до тех пор, пока не будет найдена целевая вершина или открытый список не станет пустым.

Алгоритм A\* продолжает обходить граф, пока не находит путь с минимальной стоимостью. Путь восстанавливается через ссылки на родительские вершины.

## Алгоритм ближайшего соседа

Алгоритм ближайшего соседа (Nearest Neighbor Algorithm) — это один из простейших жадных алгоритмов для решения задачи коммивояжёра [3]. Он строит маршрут, выбирая на каждом шаге ближайший не посещённый город. Рассмотрим шаги алгоритма.

**NearestNeighbor(start, intermediate\_points, goal)**

**1 посещенные ← пустое множество**

**2 маршрут ← пустой список**

**3 текущая\_точка ← start**

**4 добавь(маршрут, start)**

**5 пока есть еще непройденные промежуточные точки**

**6 ближайшая\_точка ← None**

**7 минимальное\_расстояние ← ∞**

**8 для каждого точка t из intermediate\_points**

**9 если t не в посещенные**

**10 расстояние ← A\*(текущая\_точка, t)**

**11 если расстояние < минимальное\_расстояние**

**12 ближайшая\_точка ← t**

**13 минимальное\_расстояние ← расстояние**

**14 добавь(маршрут, ближайшая\_точка)**

**15 добавь(посещенные, ближайшая\_точка)**

**16 текущая\_точка ← ближайшая\_точка**

**17 добавь(маршрут, goal)**

**18 вернуть маршрут**

Алгоритм формулируется следующим образом:

* Начало маршрута: Начинаем со стартовой точки маршрута.
* Жадный выбор: На каждом шаге выбираем ближайшую не посещённую вершину и добавляем её в маршрут.
* Завершение: Процесс продолжается до тех пор, пока не будут посещены все вершины, после чего добавляется конечная вершина.

**Преимущества**: Простота реализации и высокая скорость выполнения.

**Недостатки**: Алгоритм может дать неоптимальное решение, так как он выбирает только ближайший сосед, не учитывая долгосрочные последствия этого выбора.

## Алгоритм имитации отжига

Алгоритм имитации отжига (Simulated Annealing, SA) является мощным инструментом для поиска приближённых оптимальных решений в задаче маршрутизации, такой как задача коммивояжёра. Его основная идея заключается в том, чтобы моделировать процесс отжига, происходящий в металлургии, в котором материал подвергается нагреву и контролируемому охлаждению. В контексте задачи оптимизации имитация отжига позволяет искать глобальный минимум, избегая остановки в локальных минимумах [6]. Рассмотрим шаги алгоритма.

**SimulatedAnnealing(start, intermediate\_points, goal, initial\_temp, cooling\_rate, iterations):**

**1 текущий\_маршрут ← [start] + intermediate\_points + [goal]**

**2 текущая\_стоимость ← calculate\_total\_cost(текущий\_маршрут)**

**3 лучший\_маршрут ← текущий\_маршрут**

**4 лучшая\_стоимость ← текущая\_стоимость**

**5 температура ← initial\_temp**

**6**

**7 для i от 1 до iterations:**

**8 соседний\_маршрут ← generate\_neighbor(текущий\_маршрут)**

**9 соседняя\_стоимость ← calculate\_total\_cost(соседний\_маршрут)**

**10 delta\_стоимость ← соседняя\_стоимость - текущая\_стоимость**

**11**

**12 если delta\_стоимость < 0:**

**13 текущий\_маршрут ← соседний\_маршрут**

**14 текущая\_стоимость ← соседняя\_стоимость**

**15 иначе:**

**16 вероятность ← exp(-delta\_стоимость / температура)**

**17 if random() < вероятность:**

**18 текущий\_маршрут ← соседний\_маршрут**

**19 текущая\_стоимость ← соседняя\_стоимость**

**20**

**21 если текущая\_стоимость < лучшая\_стоимость:**

**22 лучший\_маршрут ← текущий\_маршрут**

**23 лучшая\_стоимость ← текущая\_стоимость**

**24**

**25 температура ← температура \* cooling\_rate**

**26**

**27 если температура < минимальная температура:**

**28 прервать**

**29**

**30 вернуть лучший\_маршрут**

* **Инициализация решения**: Алгоритм начинается с создания начального маршрута, который состоит из стартовой точки, всех промежуточных точек и конечной точки. Данный маршрут может быть сгенерирован случайным образом.
* **Генерация соседнего решения**: На каждом шаге алгоритм генерирует соседний маршрут путём случайного изменения текущего. В случае задачи маршрутизации это может быть перестановка двух промежуточных точек маршрута. Такое изменение называется "соседним решением".
* **Оценка стоимости нового маршрута**: После генерации соседнего маршрута вычисляется его стоимость. Это делается путём суммирования стоимости всех сегментов пути, где стоимость каждого сегмента зависит от расстояния между двумя точками и характеристик маршрута, таких как тип местности.
* **Приём нового решения**: Если стоимость нового маршрута меньше стоимости текущего маршрута, новое решение принимается без условий. В противном случае, если новое решение хуже, то оно принимается с определённой вероятностью, которая зависит от разницы в стоимости и текущей температуры. Эта вероятность снижается с уменьшением температуры.
* **Температурное охлаждение**: Температура, которая изначально установлена на высокое значение, постепенно уменьшается по мере выполнения алгоритма. Это отражает процесс охлаждения в реальной металлургии: на ранних этапах температура высокая, и алгоритм более склонен к принятию ухудшающих решений, что помогает избежать локальных минимумов. С понижением температуры вероятность принятия худшего решения становится меньше, и алгоритм начинает конвергировать к оптимальному решению.
* **Обновление лучшего решения**: Каждый раз, когда найдено лучшее решение, оно сохраняется. Алгоритм продолжает свою работу до тех пор, пока температура не станет слишком низкой или пока не будет достигнуто максимальное количество итераций.
* **Завершение алгоритма**: Когда температура опускается ниже заранее установленного порога или достигнут лимит итераций, алгоритм завершает выполнение. Результатом будет оптимальный или близкий к оптимальному маршрут с минимальной стоимостью.

**Алгоритм имитации отжига обладает рядом преимуществ:**

* Способность избегать локальных минимумов, что критично в сложных задачах с большим числом переменных.
* Хорошая гибкость в применении к различным типам задач маршрутизации, благодаря возможности настройки температуры и коэффициента охлаждения.

**Однако, у алгоритма есть и недостатки:**

* Отсутствие гарантии нахождения глобального оптимума, особенно если температура понижается слишком быстро или алгоритм не имеет достаточного времени для поиска.
* Алгоритм может потребовать значительных вычислительных ресурсов при большом количестве итераций и сложных маршрутах.

## Генетический алгоритм

Генетический алгоритм является эвристическим методом оптимизации, который использует концепцию естественного отбора и эволюции для поиска решения. Этот алгоритм применяется в задачах, где требуется найти оптимальное или близкое к оптимальному решение среди множества возможных вариантов. В данном случае, генетический алгоритм используется для решения задачи маршрутизации транспортного средства VRP, которая включает в себя нахождение оптимальных маршрутов для транспортных средств с учётом множества промежуточных точек, а также начальной и конечной точек [2].

В генетическом алгоритме решение задачи VRP представляется в виде **хромосомы** — последовательности точек, через которые транспортное средство должно пройти. Каждая хромосома в популяции — это маршрут, который начинается и заканчивается в определённых точках. Внутри популяции генерируются случайные маршруты, которые будут эволюционировать на протяжении нескольких поколений.

Рассмотрим шаги алгоритма.

**GeneticAlgorithmRouting (start, intermediate\_points, goal, terrain\_map, population\_size, generations, mutation\_rate, tournament\_size)**

**1 population ← случайная популяция маршрутов (каждый маршрут — это последовательность промежуточных точек)**

**2 best\_solution ← None**

**3 best\_length ← бесконечность**

**4 для поколения от 1 до generations:**

**5 оцененные\_популяции ← пустой список**

**6 для каждого маршрута в population:**

**7 полный\_маршрут ← [start] + маршрут + [goal]**

**8 общая\_длина ← 0**

**9 для каждого отрезка пути в полном маршруте:**

**10 длина\_отрезка ← A\*(точка1, точка2)**

**11 общая\_длина += длина\_отрезка**

**12 добавь(в оцененные\_популяции, (маршрут, общая\_длина))**

**13 отсортировать population по длине маршрута**

**14 лучший\_маршрут ← первый маршрут из отсортированной population**

**15 если длина лучшего маршрута меньше лучшей найденной длины:**

**16 обновить best\_length и best\_solution**

**17 лучшие\_маршруты ← первые половину population**

**18 new\_population ← лучшие\_маршруты**

**19 пока длина new\_population < population\_size:**

**20 родитель1 ← tournament\_selection(лучшие\_маршруты)**

**21 родитель2 ← tournament\_selection(лучшие\_маршруты)**

**22 потомок ← pmx\_crossover(родитель1, родитель2)**

**23 если случайное\_число < mutation\_rate:**

**24 потомок ← мутация(потомок)**

**25 если маршрут валиден:**

**26 добавь(в new\_population, потомок)**

**27 population ← new\_population**

**28 вернуть best\_solution**

**Инициализация популяции:**

Начальная популяция генерируется случайным образом. Каждый маршрут в популяции представляет собой хромосому — это последовательность промежуточных точек, которая будет эволюционировать на протяжении нескольких поколений.

**Оценка решений с помощью функции приспособленности:**

Каждое решение оценивается с помощью функции приспособленности. Для задачи VRP эта функция обычно зависит от общей длины маршрута. Чем короче маршрут и меньше затраты, тем лучше решение.

**Выбор лучших решений:**

После оценки всех маршрутов в популяции выбираются лучшие решения для дальнейшего использования. Это можно сделать с помощью турнирного отбора, где случайно выбираются несколько маршрутов, и среди них выбираются два лучших. Эти два маршрута будут служить родителями для следующего поколения.

**Операции скрещивания и мутации:**

Скрещивание (Crossover): После отбора родителей применяется операция скрещивания, которая комбинирует части двух маршрутов (родителей) для создания нового маршрута (потомка) [2]. Один из популярных методов скрещивания — это PMX (Partially Mapped Crossover), при котором наследуется часть маршрута от обоих родителей (рис. 1).

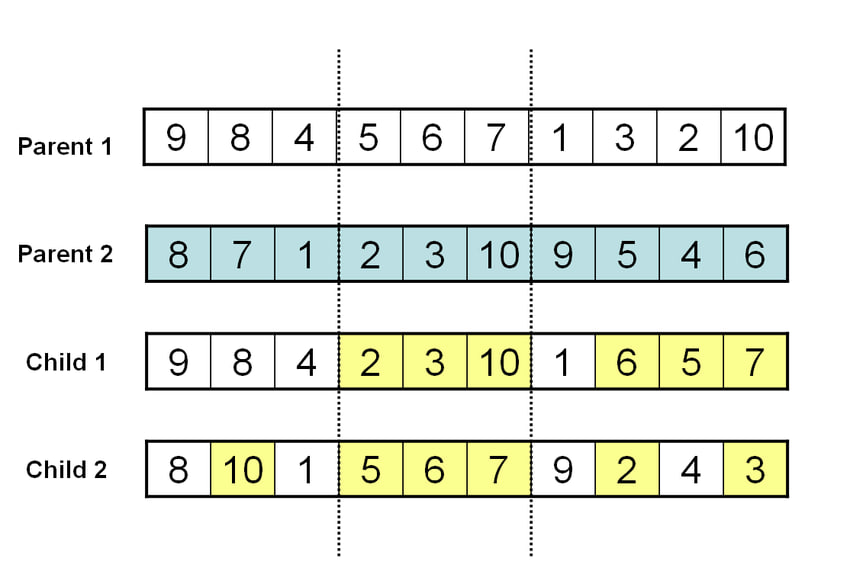


Рисунок 1. Метод скрещивания PMX

Мутация: Чтобы поддерживать разнообразие в популяции и предотвратить застой в поиске, применяется операция мутации. Мутация может включать случайное изменение маршрута, например, перестановку двух случайных промежуточных точек, инвертирование или перемешивание части маршрута.

**Эволюция популяции:**

После того как новые потомки созданы, они добавляются в новую популяцию. Эта популяция используется для следующего поколения. Таким образом, популяция эволюционирует, и с каждым поколением алгоритм улучшает маршруты, выбирая более приспособленные решения и отбрасывая менее эффективные.

**Возврат лучшего решения:**

После завершения всех поколений или выполнения условия остановки, генетический алгоритм возвращает лучшее найденное решение — маршрут, который минимизирует затраты или удовлетворяет всем ограничениям задачи.

Алгоритм имеет несколько важных преимуществ, но также и ограничения, которые необходимо учитывать при его применении.

**Плюсы генетического алгоритма**

* Глобальный поиск: в отличие от традиционных методов оптимизации, генетический алгоритм не ограничивается поиском в локальном минимуме. Благодаря случайности, мутациям и разнообразию в популяции, алгоритм способен исследовать большое пространство решений, что увеличивает вероятность нахождения глобального оптимума или решения, близкого к оптимальному.

**Минусы генетического алгоритма**

* Высокие вычислительные затраты: генетический алгоритм может требовать значительных вычислительных ресурсов, особенно при работе с большими популяциями и большим числом поколений. Для крупных задач VRP это может привести к длительным вычислениям, особенно если используются сложные функции оценки или ограничения.
* Неопределенность решения: генетический алгоритм не гарантирует нахождение глобального оптимума, поскольку его результаты зависят от начальной популяции, параметров алгоритма и случайности в процессе мутации и скрещивания.
* Зависимость от качества начальной популяции: качество начальной популяции может сильно повлиять на эффективность алгоритма. Если начальная популяция плохо представлена, алгоритм может затратить много времени на поиск более эффективных решений, что снижает общую производительность.

1. **Практическая часть**

## Общий вид программы

Программа представляет собой реализованную с помощью языка программирования Python систему для реализации и сравнения нескольких алгоритмов маршрутизации транспортных средств с учетом рельефа местности. Она использует различные методы поиска оптимальных маршрутов, такие как A\*, алгоритм ближайшего соседа, генетический алгоритм, метод симулированного отжига и метод грубой силы. Программа также генерирует случайные точки маршрута и визуализирует полученные пути с помощью библиотеки Matplotlib.

Основной функционал программы включает в себя следующие этапы:

* Генерация карты с учетом рельефа местности с помощью функции generate\_map. Она принимает параметры шума для создания карты высот (рис. 2.а), используя для этого шум Перлина из python библиотеки noise.

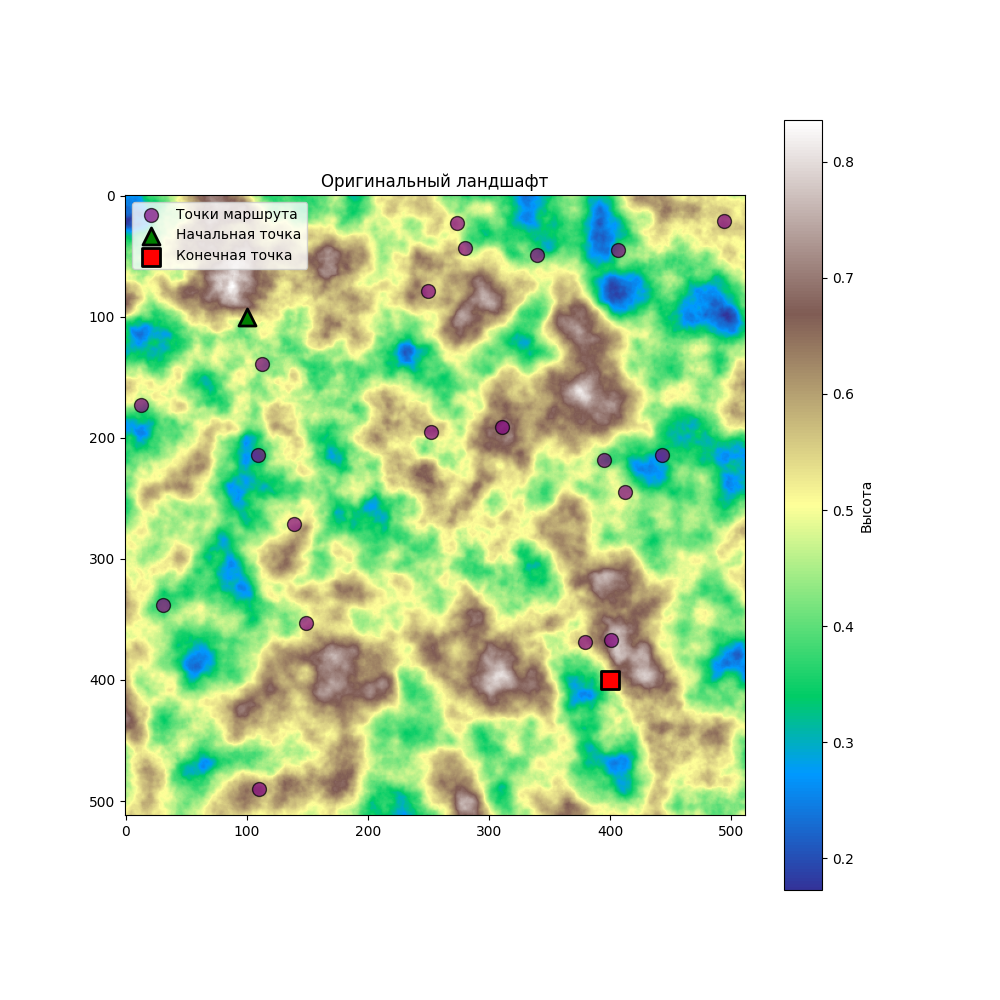
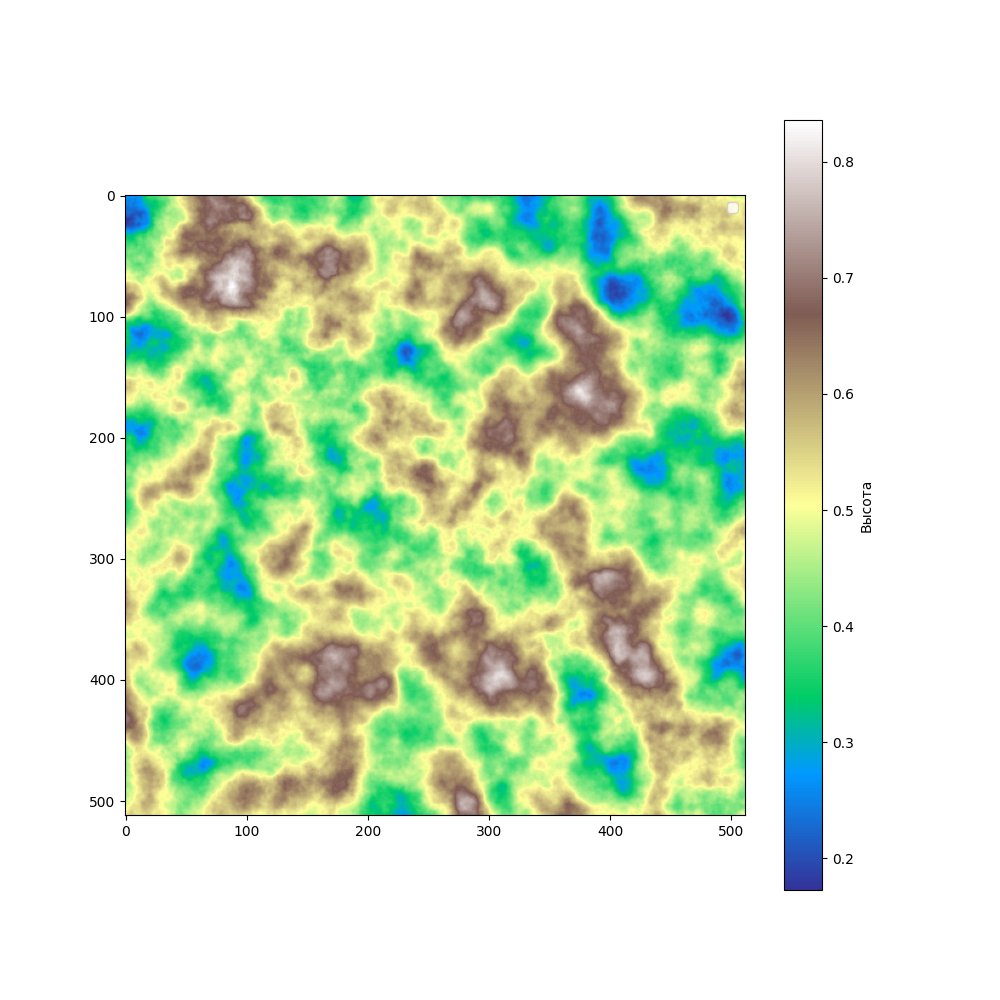


Рисунок 2. а) Сгенерированная карта высот

б) Ландшафт с расставленными точками

* Расстановка случайных или записанных точек начала, конца и некоторого числа промежуточных точек (рис. 2.б).
* Планирование маршрута с использованием различных алгоритмов, таких как A\*, генетический алгоритм, метод грубой силы, метод ближайшего соседа и симулированный отжиг. Для каждого алгоритма есть соответствующие функции, которые принимают на вход начальную и конечную точку маршрута, а также список промежуточных точек.
* Визуализация результата с помощью библиотеки Matplotlib. Результаты отображаются на карте, где проложенные маршруты выделяются красным цветом, а также выводится информация о длине маршрута для каждого алгоритма.

Один из ключевых аспектов реализации — **кэширование путей и их длин**, что существенно повышает эффективность работы программы, особенно при многократном использовании одинаковых путей между точками.

Кэширование путей позволяет избежать повторных вычислений для путей между одинаковыми точками, что критически важно при использовании таких методов, как алгоритм грубой силы, где каждый маршрут строится через несколько промежуточных точек. Для каждого уникального пути между двумя точками сохраняется результат в кэше. Когда нужно снова вычислить путь между этими же точками, программа просто извлекает уже готовый результат из кэша, что значительно сокращает время вычислений.

## Алгоритм А\*

В рамках проекта алгоритм A\* используется как базовый метод для нахождения пути между двумя точками. Он служит основой для реализации других алгоритмов.

В данной реализации алгоритма A\* используется Python с библиотеками, такими как NumPy и Matplotlib, для обработки данных и визуализации результатов.

Классы и функции:

* Node: класс, представляющий узел графа (рис. 3). Каждый узел хранит свою позицию, стоимость пути до него (g-стоимость), эвристическую оценку (h-стоимость) и общую стоимость (f-стоимость). Также в классе содержится ссылка на родительский узел для восстановления пути.

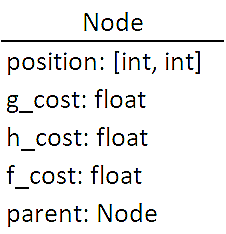


Рисунок 3. Класс Node

* a\_star\_path (): основная функция для поиска пути. В ней используется очередь с приоритетами для поиска узлов с минимальной f-стоимостью. В процессе работы она вычисляет g- и h-стоимости для каждого узла и его соседей.
* heuristic (): функция для вычисления эвристического расстояния между двумя точками, в нашей реализации используется манхэттенское расстояние, равное сумме модулей разностей координат точек (рис. 4).

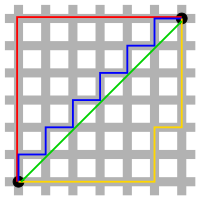


Рисунок 4. Манхэттенское расстояние

* calculate\_path\_length (): функция для вычисления общей длины пути с учётом рельефа (разница высот между соседними точками).
* calculate\_height (): вспомогательная функция для вычисления высотной разницы между двумя точками, которая учитывает рельеф местности, опираясь на коэффициент height\_weight.

Программа выполняет поиск пути с использованием алгоритма A\* на карте, где:

* Каждая точка карты (ячейка) представляет собой узел графа.
* Рёбра между узлами соответствуют возможным путям перемещения, стоимость которых зависит от высоты местности (рельефа).

В результате выполнения алгоритма A\* программа выводит найденный путь от начальной точки до целевой, отображая его на карте (рис. 5). Программа также вычисляет и выводит длину пути с учётом рельефа (включая изменения высот) и эвристической стоимости.

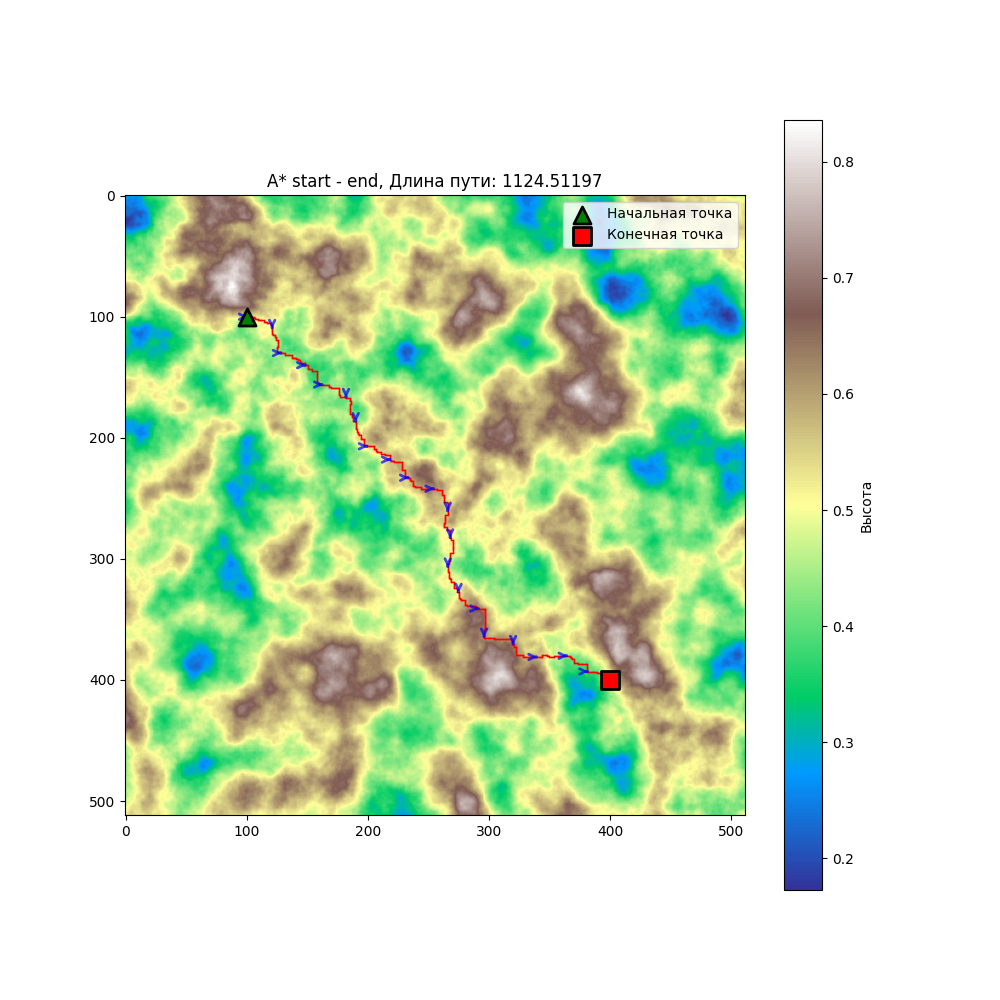


Рисунок 5. Рассчитанный путь А\* между двумя точками

Если включена опция отображения посещённых точек, программа выводит на экран все узлы, которые были обработаны на пути поиска (рис.6).

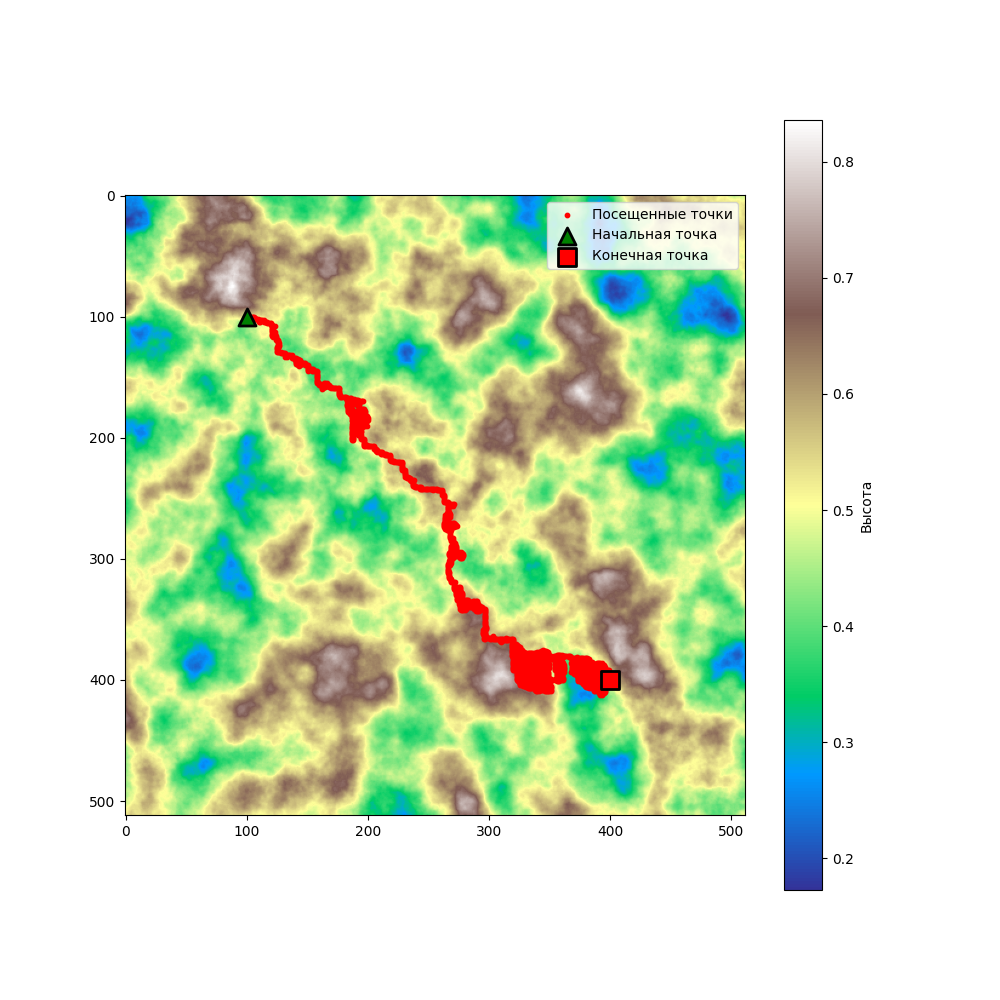


Рисунок 6. Посещённые во время работы алгоритма точки

## Алгоритм грубой силы

Алгоритм грубой силы, также известный как метод полного перебора, представляет собой один из самых простых, но неэффективных способов нахождения оптимального маршрута между несколькими точками. Этот метод заключается в переборе всех возможных маршрутов, что позволяет гарантировать нахождение самого короткого пути, но требует значительных вычислительных ресурсов, особенно при увеличении количества точек.

В рамках нашего проекта, данный алгоритм был использован для нахождения пути с учётом рельефа местности. Несмотря на свою высокую вычислительную стоимость, он гарантирует получение самого короткого пути, что является его основным преимуществом.

**Представление маршрута**: Алгоритм начинается с генерации всех возможных перестановок промежуточных точек — permutations библиотеки itertools. Для каждой перестановки промежуточных точек строится маршрут, который включает начальную точку, переставленные промежуточные точки и конечную точку.

**Оптимальный маршрут**: Для каждого маршрута вычисляется его общая стоимость. Алгоритм вычисляет все возможные маршруты и выбирает тот, который имеет минимальную стоимость. Этот маршрут и будет считаться оптимальным (рис. 7).

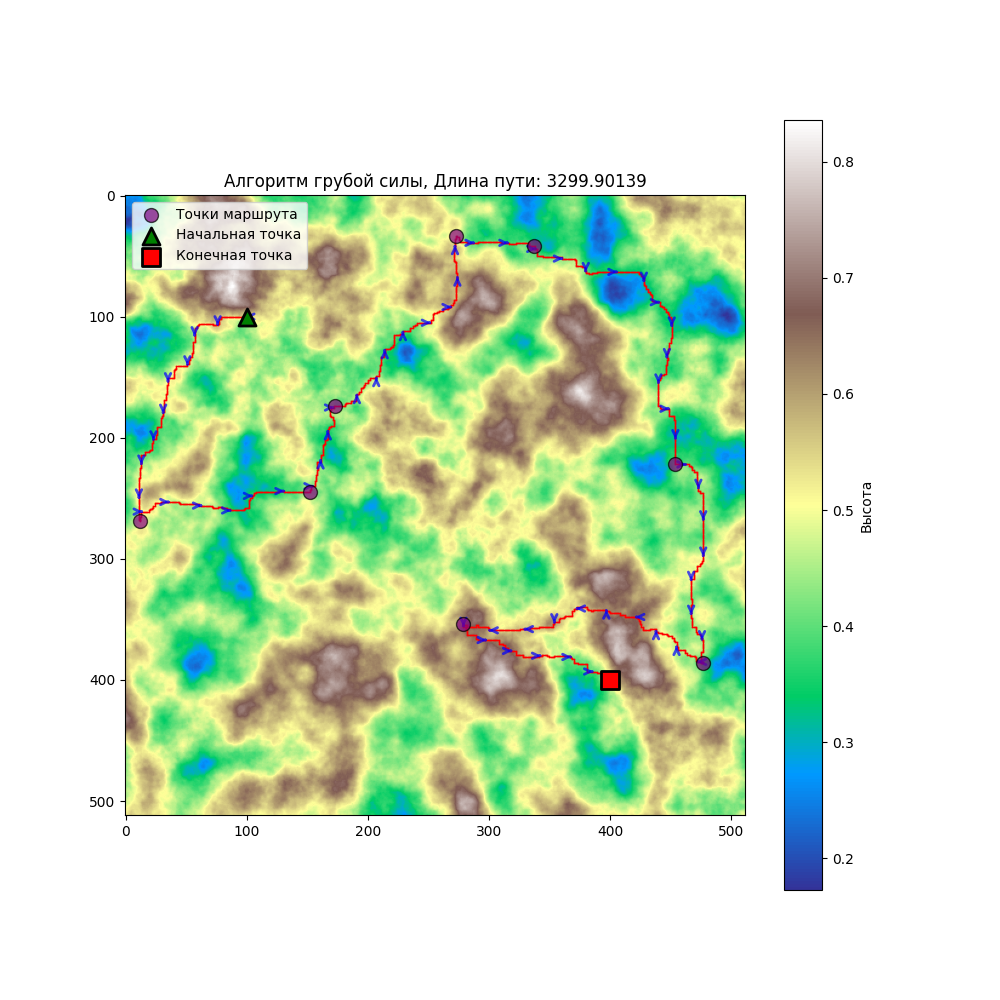


Рисунок 7. Путь, просчитанный алгоритмом грубой силы для 8 промежуточных точек

**Подсчёт шагов**: в процессе работы алгоритм подсчитывает количество перестановок маршрутов и количество проверок путей. Эти данные помогают оценить эффективность работы алгоритма. Количество реальных вычислений путей также фиксируется, что позволяет анализировать эффективность использования кэша.

Рассчитанные данные для 8 промежуточных точек:

* Длина пути (Алгоритм грубой силы): 3299.90139
* Количество перестановок (маршрутов): 40320
* Количество проверок путей: 362880
* Количество реальных вычислений путей: 72

Преимуществом алгоритма грубой силы является гарантированное нахождение оптимального маршрута. Полный перебор всех вариантов позволяет исключить возможность получения локальных оптимумов решений.

Недостатком алгоритма является высокая вычислительная сложность: Алгоритм требует вычисления всех возможных маршрутов, что приводит к экспоненциальному росту времени работы с увеличением количества точек. Количество маршрутов растёт как факториал от количества промежуточных точек. Ограничение на количество точек: Алгоритм эффективно работает только с малым числом точек (не более 10), поскольку для более сложных задач его вычислительные затраты становятся неприемлемо большими.

## Алгоритм ближайшего соседа

Алгоритм ближайшего соседа — это жадный алгоритм для поиска маршрута, который на каждом шаге выбирает ближайшую к текущей точку среди оставшихся не посещённых. Этот метод прост в реализации и хорошо подходит для решения задачи маршрутизации с множеством промежуточных точек, однако он не всегда даёт оптимальный путь.

**Инициализация точек и маршрута**: Алгоритм начинается с инициализации списка не посещённых точек, добавления начальной точки в маршрут и подготовки к поиску ближайших точек. На каждом шаге вычисляется стоимость пути до всех оставшихся не посещённых точек, после чего выбирается ближайшая точка.

**Построение маршрута**: Путь от текущей точки до ближайшей добавляется в общий маршрут, и алгоритм переходит к следующей точке. Этот процесс повторяется, пока все промежуточные точки не будут добавлены в маршрут.

**Добавление пути к конечной точке**: После того как все промежуточные точки посещены, алгоритм вычисляет путь к конечной точке и добавляет его в конечный маршрут (рис. 8).

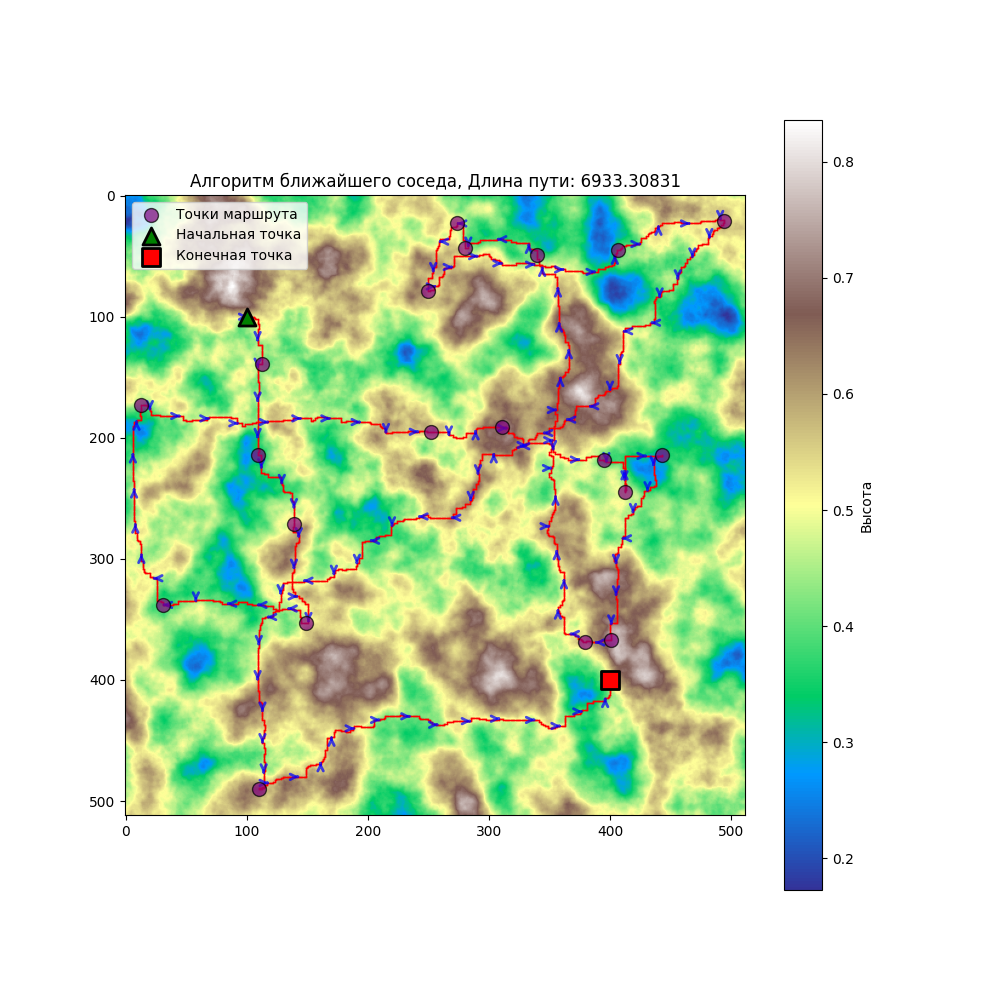
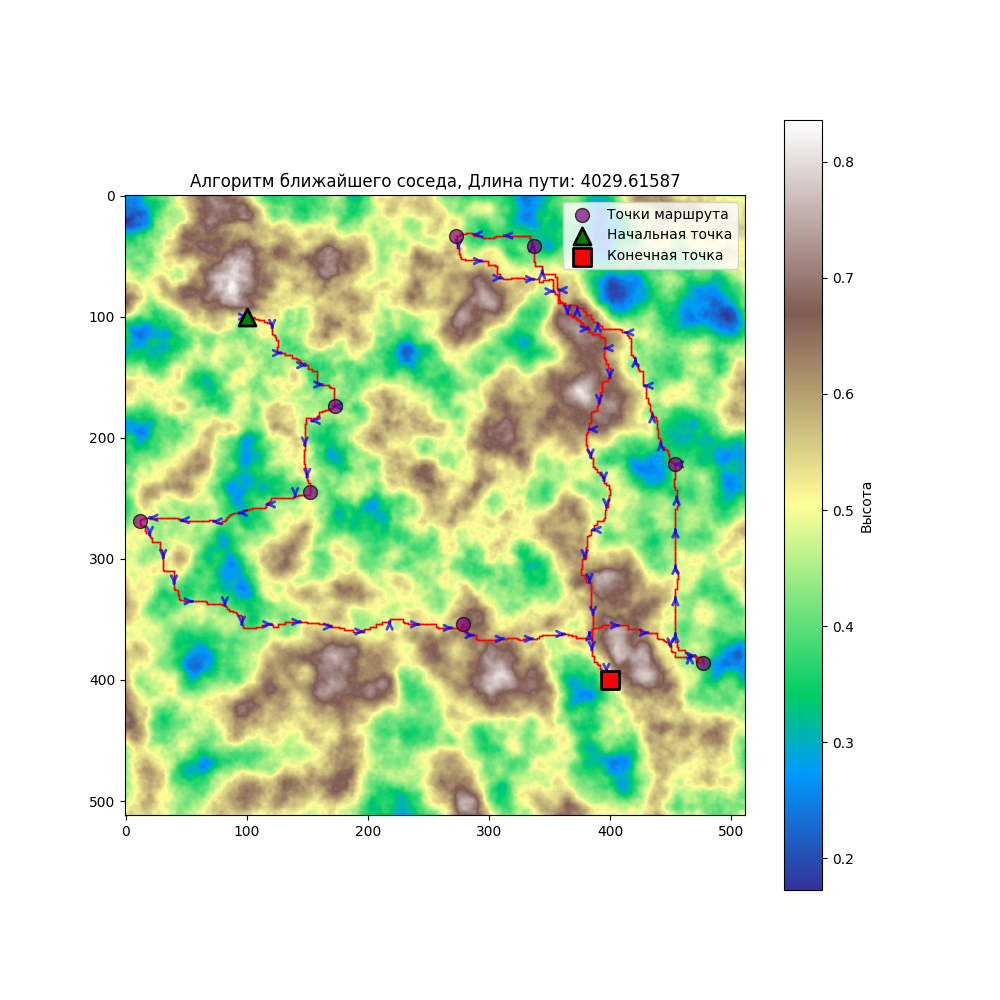


Рисунок 8. Путь, просчитанный алгоритмом ближайшего соседа для, а) 8 промежуточных точек, б) 20 промежуточных точек

**Подсчёт шагов**: в конце алгоритм выводит статистику о количестве проверок путей и количестве реальных вычислений, которые были выполнены с использованием кэшированных данных.

Рассчитанные данные для 8 промежуточных точек:

* Количество проверок путей: 45
* Количество реальных вычислений путей: 37
* Длина пути (Алгоритм ближайшего соседа): 4029.61587

Рассчитанные данные для 20 промежуточных точек:

* Количество проверок путей: 231
* Количество реальных вычислений путей: 211
* Длина пути (Алгоритм ближайшего соседа): 6933.30831

**Анализ**: Если сравнивать алгоритм ближайшего соседа с методом грубой силы, то можно отметить, хотя ближайший сосед часто находит более длинный путь (рис.8а), его эффективность в плане количества проверок и вычислений значительно выше, что делает его более подходящим для задач с большим числом точек.

## Алгоритм имитации отжига

Алгоритм имитации отжига (Simulated Annealing) основывается на принципах термодинамики и физики, где система проходит через стадии охлаждения, чтобы прийти к глобальному минимуму, начиная с высокого "энергетического" состояния и постепенно снижая его, принимая решения о переходе к худшим состояниям с вероятностью, зависящей от температуры. В контексте маршрутизации, алгоритм используется для поиска кратчайшего пути с учетом рельефа через последовательные итерации изменения маршрута.

Функции:

* generate\_neighbor (): Функция генерирует соседнее решение, изменяя порядок промежуточных точек маршрута. Сначала выбираются два случайных индекса, не включая начальную и конечную точку, затем местами меняются промежуточные точки на этих позициях.
* calculate\_total\_cost (): Рассчитывает общую стоимость маршрута, суммируя длины всех его сегментов. Важно отметить, что используются кэшированные значения длины путей, что ускоряет вычисления, особенно при многократных расчетах для одних и тех же точек маршрута.
* simulated\_annealing\_routing (): Основная функция алгоритма имитации отжига. Она начинает с генерации случайного начального маршрута, вычисляет его стоимость и начинает процесс охлаждения.

**Инициализация решения**: Алгоритм начинается с генерации случайного начального решения, которое состоит из последовательности точек, включающих начальную точку, все промежуточные точки маршрута и конечную точку. Для каждой последовательности точек алгоритм вычисляет её стоимость.

**Генерация соседнего решения:** Алгоритм пытается улучшить текущее решение путём случайных изменений. Генерация соседнего решения включает изменение порядка промежуточных точек (не затрагивая начальную и конечную точки). Это достигается случайной перестановкой двух промежуточных точек.

**Оценка нового решения:** После генерации нового маршрута алгоритм оценивает его стоимость. Если новое решение имеет меньшую стоимость, оно принимается. Если новое решение длиннее, алгоритм может принять его с некоторой вероятностью, которая зависит от разницы стоимости и температуры.

**Температура и вероятность принятия:** Вероятность принятия худшего решения с течением времени снижается, так как температура постепенно уменьшается. Температура на каждом шаге её значение уменьшается с помощью множителя охлаждения (cooling\_rate). Это позволяет алгоритму на начальных этапах выполнять большие изменения (поиск в более широком пространстве решений), а к финалу сужать поиск, чтобы найти оптимальное решение.

**Завершение работы**: Алгоритм завершает свою работу, когда температура падает до достаточно низкого значения, или если выполнено заданное количество итераций. Результатом работы является последовательность точек, которая представляет собой оптимальный близкий к оптимальному маршрут (рис.9).

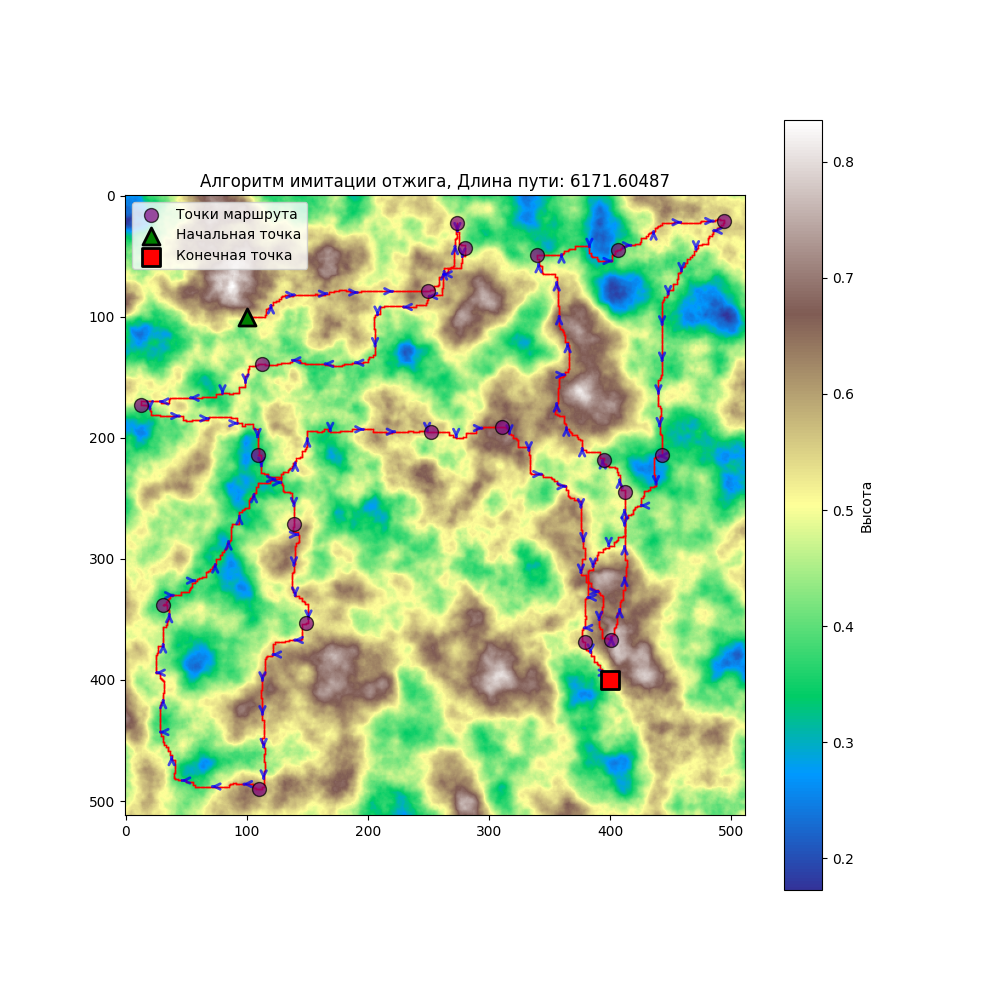


Рисунок 9. Путь, просчитанный алгоритмом имитации отжига для 20 промежуточных точек



Рисунок 10. График зависимости длины маршрута от итерации в алгоритме имитации отжига

Рассчитанные данные для 20 промежуточных точек:

* Количество проверок путей: 21021
* Количество реальных вычислений путей: 419
* Длина пути (Алгоритм имитации отжига): 6171.60487

Приведенный путь (рис.9) был построен при начальной температуре initial\_temp = 2000, темпе охлаждения cooling\_rate = 0,995. Было проведено 1000 итераций.

**Анализ**: Алгоритм имитации отжига нашел более короткий по сравнению с алгоритмом ближайшего соседа маршрут, но его работа сильно зависит от входных значений и случайности.

**Преимущества алгоритма имитации отжига:**

* Поиск глобального оптимума: в отличие от некоторых других методов, таких как ближайший сосед, алгоритм имитации отжига может эффективно искать глобальный минимум. Это достигается за счёт случайного принятия худших решений с определенной вероятностью, что помогает избежать остановки в локальных минимумах.
* Постепенное улучшение решения: Алгоритм позволяет постепенно улучшать решение, начиная с неоптимального, с понижением температуры, что способствует поиску более точного решения с течением времени.

**Недостатки алгоритма имитации отжига:**

* Зависимость от случайности: Алгоритм сильно зависит от случайных изменений в решении. На каждом шаге случайно выбираются новые решения, что может привести к нестабильности результатов. Иногда алгоритм может найти плохое решение, если случайные изменения не приведут к улучшению.
* Потребность в настройке параметров: Один из основных недостатков алгоритма — это необходимость аккуратно настраивать параметры, такие как начальная температура, коэффициент охлаждения и количество итераций. Неправильная настройка этих параметров может значительно повлиять на эффективность и точность решения (рис.11). Например, слишком быстрое охлаждение может привести к неадекватному поиску оптимума, а слишком медленное — к неоправданно долгому времени выполнения.

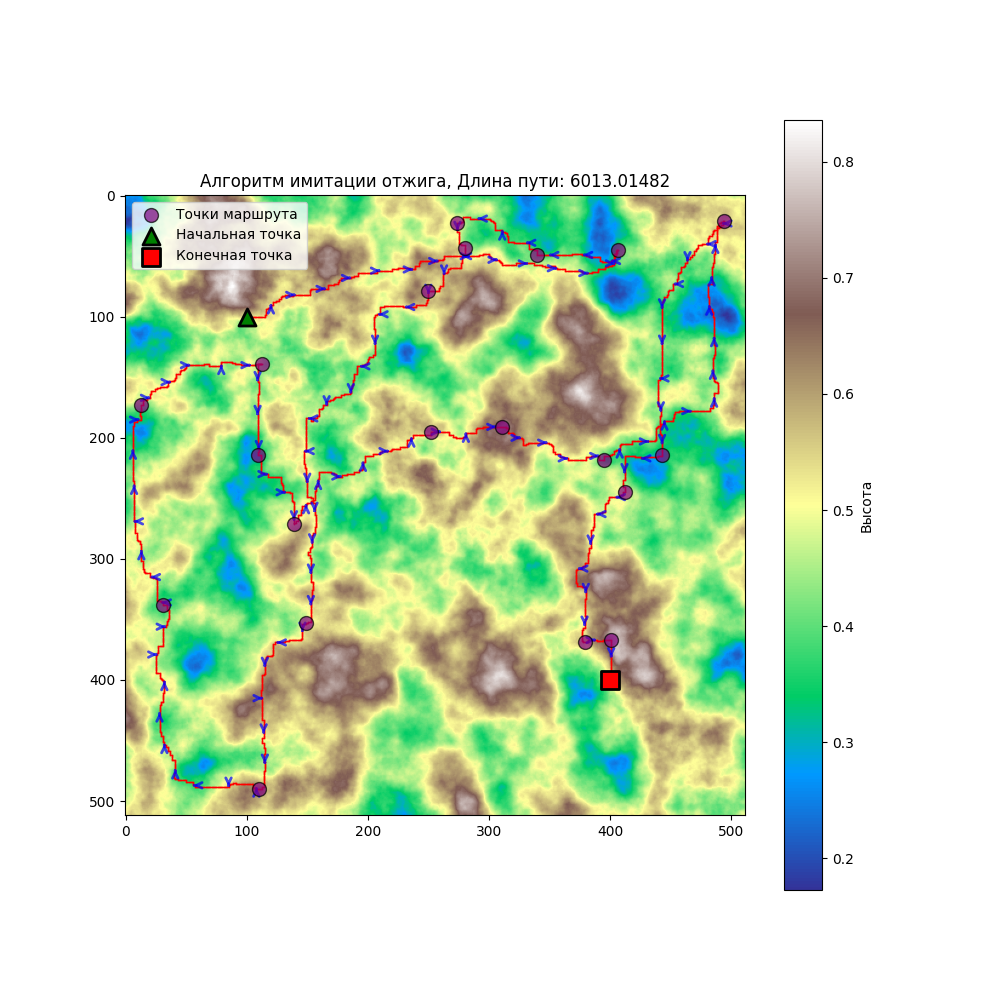


Рисунок 11. Пути, просчитанные имитацией отжига при темпе охлаждения, а) 0,990 б) 0,997

## Генетический алгоритм

Генетический алгоритм — это эвристический метод оптимизации, основанный на принципах естественного отбора и эволюции. Алгоритм моделирует процесс естественного отбора, где особи, представляющие потенциальные решения задачи, подвергаются мутациям, скрещиваниями и отбору для улучшения популяции на каждом шаге. В контексте маршрутизации, генетический алгоритм используется для поиска оптимального маршрута, уменьшая длину пути, проходя через множество промежуточных точек с учетом рельефа местности.

Функции:

* genetic\_algorithm\_routing(): Реализует основной цикл генетического алгоритма: инициализация популяции, отбор родителей, применение скрещивания и мутаций, создание новой популяции, вычисление качества решения и проверка условий остановки.
* tournament\_selection(): Производится турнирный отбор, где несколько случайных решений оцениваются по качеству, и из них выбираются лучшие для дальнейшего скрещивания.
* pmx\_crossover(): Генерирует два новых маршрута (потомков) путем частичного обмена частями маршрутов двух родителей, при этом гарантируется отсутствие дублирования точек в потомках (рис. 12).

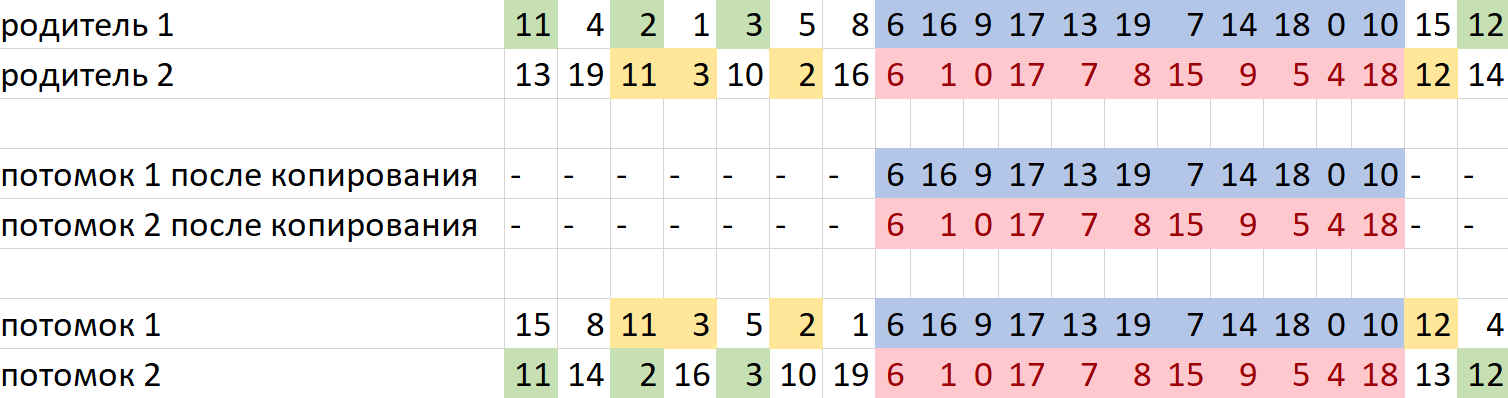


Рисунок 12. Пример работы скрещивания PMX

* inverse\_mutation(): Переворачивает порядок промежуточных точек на случайном участке маршрута (рис. 13).

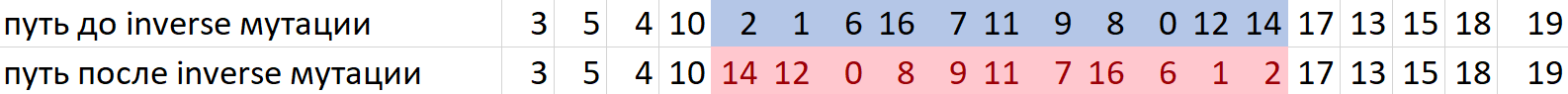


Рисунок 13. Пример работы inverse мутации

* scramble\_mutation(): Перемешивает порядок промежуточных точек на случайном участке маршрута (рис. 14).

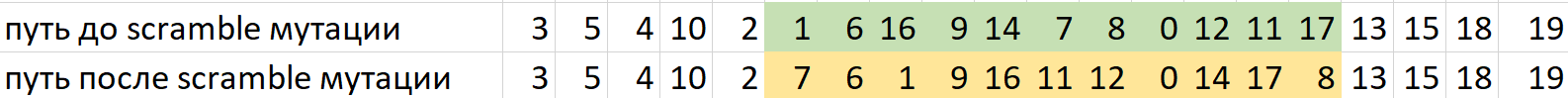


Рисунок 14. Пример работы scramble мутации

* swap\_mutation(): Меняет местами две случайные промежуточные точки маршрута (рис. 15).



Рисунок 15. Пример работы swap мутации

* check\_valid\_route(): Проверка на валидность маршрута, исключая маршруты с дублирующимися точками или неполные маршруты.

**Инициализация решения:** Алгоритм начинается с создания случайной начальной популяции маршрутов задаваемого размера population\_size, где каждый маршрут представляет собой последовательность промежуточных точек. На каждом маршруте оценивается длина маршрута, путем добавления слева к нему начальной и справа конечной точки.

**Отбор и создание новой популяции:** Популяция обновляется путем выбора лучших решений из текущего поколения с использованием турнирного отбора: отбирается задаваемое параметром tournament\_size количество особей, далее создаются новые особи, которые добавляются в следующую популяцию (рис. 16).

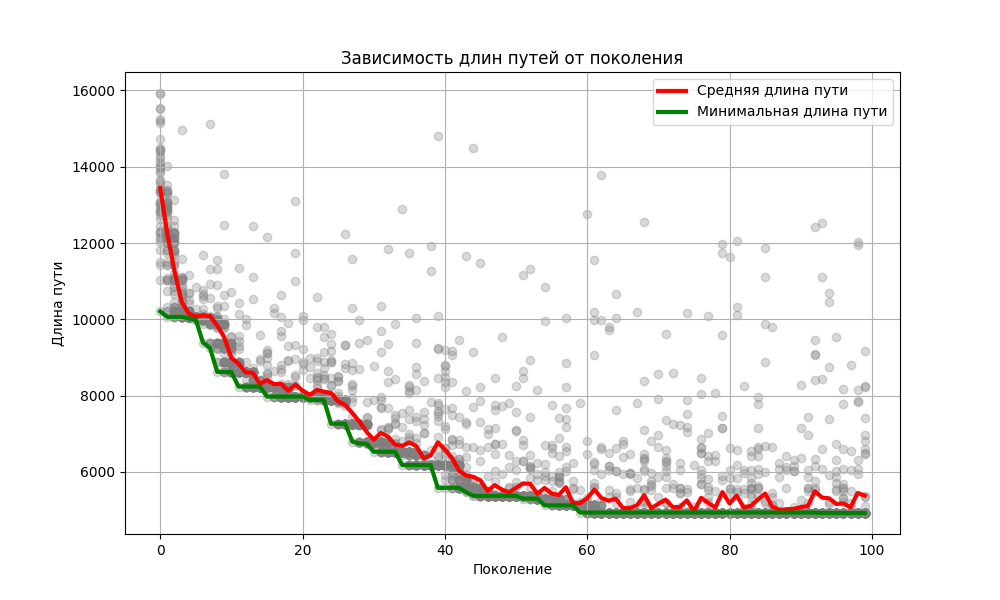


Рисунок 16. Зависимость длины путей каждой особи в каждом поколении

**Генерация нового решения (скрещивание и мутация):** Каждое новое решение получается путем выбора двух родителей из текущей популяции, скрещивания их с использованием метода PMX и применения мутации. Мутации могут произойти с задаваемой вероятностью mutation\_rate, вид мутации выбирается случайно, в первых поколениях с равной вероятностью выбирается инвертирование, перемешивание или перестановка, ближе к концу для увеличения разнообразия используются более агрессивные мутации инвертирования или перемешивания.

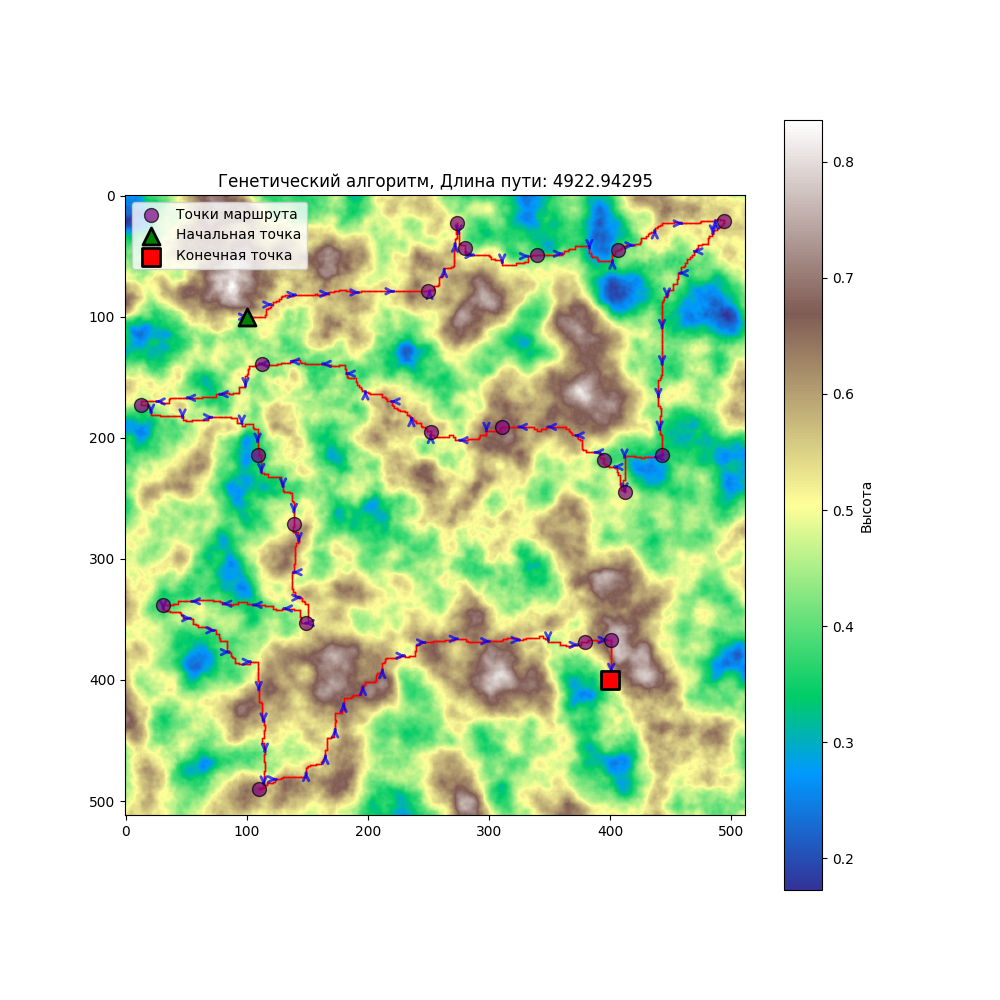


Рисунок 17. Путь, просчитанный генетическим алгоритмом для 20 промежуточных точек

**Завершение работы:** Алгоритм продолжается до тех пор, пока не будет выполнено заданное количество поколений generation. Результатом работы алгоритма является последовательность точек, которая представляет собой оптимальный маршрут (рис. 17).

Рассчитанные данные для 20 промежуточных точек:

* Количество проверок путей: 345800
* Количество реальных вычислений путей: 37454
* Длина пути (Генетический алгоритм): 4922.94295

Приведенный путь (рис.17) был построен при размере популяций population\_size = 50, вероятности мутации mutation\_rate = 0,3 и начальном размере турнира tournament\_size = 10. Было просчитано 100 поколений.

**Анализ**: Количество проверенных и реально просчитанных путей значительно больше чем в алгоритме ближайшего соседа и имитации отжига, но в результате генетического алгоритма был найден более короткий маршрут.

**Преимущества генетического алгоритма:**

* Поиск глобального оптимума: Генетический алгоритм может эффективно искать глобальный минимум, избегая остановки в локальных оптимумах благодаря случайным мутациям и скрещиваниям. Процесс скрещивания и мутации позволяет генерировать разнообразные решения, что помогает избежать ранней сходимости к неоптимальным решениям.

**Недостатки генетического алгоритма:**

* Вычислительные затраты: Из-за необходимости многократного выполнения скрещиваний, мутаций и оценки качества решений, генетический алгоритм может быть вычислительно затратным, особенно для больших пространств поиска.
* Сложность интерпретации результатов: Из-за случайности в алгоритме результаты могут быть не предсказуемыми, что затрудняет точную оценку качества алгоритма без дополнительных метрик или экспериментов.

## Полученные результаты

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество точек** | **Использованный алгоритм** | **Количество проверок** | **Количество просчитанных путей** | **Длина вычисленного маршрута** |
| 8 | Грубой силы | 362880 | 72 | 3299.90139 |
| 8 | Ближайшего соседа | 45 | 37 | 4029.61587 |
| 20 | Ближайшего соседа | 231 | 211 | 6933.30831 |
| 20 | Имитации отжига, cooling\_rate = 0,990 | 21021 | 420 | 6013.01482 |
| 20 | Имитации отжига, cooling\_rate = 0,995 | 21021 | 419 | 6171.60487 |
| 20 | Имитации отжига, cooling\_rate = 0,997 | 21021 | 420 | 7403.65433 |
| 20 | Генетический | 345800 | 37454 | 4922.94295 |

**Заключение**

В ходе проведенной работы были изучены алгоритмы для планирования маршрутов транспортных средств с учетом рельефа местности, при помощи средств языка программирования Python и его библиотек эти алгоритмы были реализованы и был проведен сравнительный анализ.

Все задачи, поставленные в начале, были решены – реализована генерация карты высот при помощи библиотек noise и Matplotlib. Были реализованы такие алгоритмы планирования маршрута как алгоритм А\*, алгоритм грубой силы, алгоритм ближайшего соседа, имитация отжига и генетический алгоритм. Были визуализированы просчитанный путь между настраиваемым числом точек на карте с выводом длины конечного пути и просчитанных шагов.

В ходе работы были получены такие результаты:

* Точное вычисление пути методом грубой силы, хотя и является самым точным, оказалось крайне медленным и неприемлемым для реальных задач, требующих быстрой обработки данных.
* Алгоритм ближайшего соседа, хоть и не обеспечивал наилучшее решение с точки зрения точности, продемонстрировал высокую скорость работы, что делает его привлекательным для задач, где важна скорость вычислений, а не абсолютная оптимальность.
* Алгоритм симулированного отжига, будучи достаточно точным, показал зависимость от случайных факторов. Это делает его решение менее предсказуемым, а сам алгоритм — медленным, что ограничивает его использование в условиях, требующих быстрой работы.
* Генетический алгоритм, как и алгоритм симулированного отжига, также зависел от случайности, но при этом обеспечивал достаточно хорошее решение по точности. Однако, его выполнение также было медленным, что затрудняет его использование для реальных задач с большими данными.

Хотя программа предоставляет возможность для эффективного планирования и визуализации маршрутов, есть возможности для улучшения её производительности и точности.

Одно из возможных улучшений – оптимизация работы имеющихся алгоритмов или реализации новых, более совершенных методов. Также возможно нахождение более точных параметров методов, что позволит вычислять маршруты быстрее и с меньшими затратами вычислительных мощностей.

**Список литературы**

1. Белла Р., Лейк К. Алгоритмы для графов. — СПб.: Питер, 2008.
2. В М. Гавриляко Задача коммивояжера и генетические алгоритмы // Радиоэлектроника, информатика, управления. 2003. №1 (9). URL: https://cyberleninka.ru/article/n/zadacha-kommivoyazhera-i-geneticheskie-algoritmy.
3. Вигдарт Р. Алгоритмы и структуры данных. — М.: Открытые системы, 2010.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Глава 22. Элементарные алгоритмы для работы с графами // Алгоритмы: построение и анализ (второе издание). — М.: «Вильямс», 2005. — С. 622—632.
5. Лестер П. Алгоритм A\* для новичков. 2004. URL: https://masters.donntu.ru/2014/fknt/shevchenko/library/article5.htm
6. Мишин А., Савельев М. Алгоритм имитации отжига и его применение. — М.: Наука, 2015.

**Приложение А**

**Листинг программы**

**annealing.py:**

import random

import math

from typing import List, Tuple

import numpy as np

from paths import get\_path\_from\_cache\_or\_calculate, get\_path\_length\_from\_cache\_or\_calculate

from map import plot\_annealing\_data

from settings import plot\_graph\_annealing

path\_cache = {}

path\_length\_cache = {}

# Счетчики для подсчета проверок путей

path\_check\_count = 0

# Функция для генерации соседнего маршрута (изменение порядка промежуточных точек)

def generate\_neighbor(route: List[Tuple[int, int]]) -> List[Tuple[int, int]]:

new\_route = route.copy()

# Генерируем два случайных индекса для промежуточных точек (не включая начальную и конечную точку)

idx1, idx2 = random.sample(range(1, len(route) - 1), 2)

new\_route[idx1], new\_route[idx2] = new\_route[idx2], new\_route[idx1] # Меняем местами их

return new\_route

# Функция для расчета длины пути с использованием кэша

def calculate\_total\_cost(route: List[Tuple[int, int]], \_terrain\_map: np.ndarray) -> float:

global path\_check\_count

total\_path = []

total\_cost = 0.0

# Рассчитываем путь от начальной точки до первой промежуточной

for i in range(len(route) - 1):

# Проверяем, есть ли путь в кэше

path\_check\_count += 1 # Каждая проверка пути увеличивает счетчик

segment = get\_path\_from\_cache\_or\_calculate(route[i], route[i + 1], \_terrain\_map, path\_cache, path\_length\_cache)

total\_path.extend(segment) # Добавляем все точки сегмента, включая начальную

# Рассчитываем стоимость пути

total\_cost += get\_path\_length\_from\_cache\_or\_calculate(route[i], route[i + 1], \_terrain\_map, path\_cache,

path\_length\_cache)

return total\_cost

# Функция для симулированного отжига

def simulated\_annealing\_routing(\_start: Tuple[int, int], \_points: List[Tuple[int, int]], \_end: Tuple[int, int],

\_terrain\_map: np.ndarray, \_initial\_temp: float, \_cooling\_rate: float,

\_iterations: int) -> List[Tuple[int, int]]:

global path\_check\_count

# Инициализация начального решения: стартовая точка + промежуточные точки + конечная точка

current\_solution = [\_start] + \_points + [\_end]

current\_cost = calculate\_total\_cost(current\_solution, \_terrain\_map)

best\_solution = current\_solution

best\_cost = current\_cost

# Список для хранения длины пути на каждой итерации

costs = [current\_cost]

# Начальная температура

temperature = \_initial\_temp

for iteration in range(\_iterations):

# Генерация соседнего решения (меняем местами промежуточные точки)

neighbor\_solution = generate\_neighbor(current\_solution) # Генерация соседа

# Расчет стоимости нового маршрута

neighbor\_cost = calculate\_total\_cost(neighbor\_solution, \_terrain\_map)

# Вычисление изменения стоимости

delta\_cost = neighbor\_cost - current\_cost

# Если новое решение лучше, принимаем его

if delta\_cost < 0.0:

current\_solution = neighbor\_solution

current\_cost = neighbor\_cost

else:

# Если новое решение хуже, принимаем его с вероятностью exp(-delta\_cost / temperature)

acceptance\_probability = math.exp(-delta\_cost / temperature)

if random.random() < acceptance\_probability:

current\_solution = neighbor\_solution

current\_cost = neighbor\_cost

# Обновление лучшего решения, если новое лучше

if current\_cost < best\_cost:

best\_solution = current\_solution

best\_cost = current\_cost

# Понижаем скорость охлаждения на каждом шаге

if iteration % 100 == 0:

print(

f"Итерация {iteration}, Текущая стоимость: {current\_cost}, Лучшее решение: {best\_cost}, Температура: {temperature}")

# Добавляем текущую длину пути в список для графика

costs.append(current\_cost)

# Понижение температуры

temperature \*= \_cooling\_rate

# Если температура слишком мала, прекращаем итерации

if temperature < 1e-5:

break

final\_path = [\_start] # Начальная точка

for i in range(len(best\_solution) - 1):

segment = get\_path\_from\_cache\_or\_calculate(best\_solution[i], best\_solution[i + 1], \_terrain\_map, path\_cache,

path\_length\_cache)

final\_path.extend(segment)

# Выводим количество просчитанных путей и проверок

print(f"Количество проверок путей: {path\_check\_count}")

print(f"Количество реальных вычислений путей: {len(path\_length\_cache)}")

# Очищаем кэш путей после завершения маршрута

path\_cache.clear()

path\_length\_cache.clear()

# Строим график длины пути в зависимости от итерации

if plot\_graph\_annealing:

plot\_annealing\_data(costs, len(costs))

return final\_path

**brute\_force.py:**

import itertools

from typing import List, Tuple

import numpy as np

from paths import get\_path\_from\_cache\_or\_calculate, get\_path\_length\_from\_cache\_or\_calculate

path\_cache = {}

path\_length\_cache = {}

# Функция для маршрутизации методом полного перебора всех маршрутов с промежуточными точками

def brute\_force\_routing(\_start: Tuple[int, int], \_points: List[Tuple[int, int]], \_end: Tuple[int, int],

\_terrain\_map: np.ndarray) -> List[Tuple[int, int]]:

# Проверяем количество точек и предупреждаем, если их слишком много

if len(\_points) > 10:

print("Внимание! Число точек больше 10, алгоритм будет работать очень долго!")

final\_path = None

min\_cost = float('inf')

total\_permutations = 0 # Считаем количество перестановок

total\_path\_checks = 0 # Считаем количество вычислений путей

# Перебор всех перестановок

for perm in itertools.permutations(\_points):

total\_permutations += 1 # Увеличиваем счетчик перестановок

current\_path = [\_start] + list(perm) + [\_end]

total\_cost = 0

full\_path = []

# Проходим по всем сегментам маршрута

for i in range(len(current\_path) - 1):

# Для каждого сегмента пути между точками вычисляем путь с учетом ландшафта

segment\_path = get\_path\_from\_cache\_or\_calculate(current\_path[i], current\_path[i + 1], \_terrain\_map,

path\_cache, path\_length\_cache)

total\_path\_checks += 1 # Увеличиваем счетчик вычислений путей

# Добавляем сегмент пути в общий путь

if i != len(current\_path) - 2: # не добавляем конечную точку повторно

full\_path.extend(segment\_path[:-1])

else:

full\_path.extend(segment\_path) # Для последнего сегмента включаем конечную точку

# Рассчитываем стоимость сегмента

segment\_length = get\_path\_length\_from\_cache\_or\_calculate(current\_path[i], current\_path[i + 1],

\_terrain\_map, path\_cache, path\_length\_cache)

total\_cost += segment\_length

# Если найден новый минимальный путь, сохраняем его

if total\_cost < min\_cost:

min\_cost = total\_cost

final\_path = full\_path

# Выводим количество шагов

print(f"Количество перестановок (маршрутов): {total\_permutations}")

print(f"Количество проверок путей: {total\_path\_checks}")

print(f"Количество реальных вычислений путей: {len(path\_length\_cache)}")

# Очищаем кэш путей после завершения маршрута

path\_cache.clear()

path\_length\_cache.clear()

return final\_path

**genetic.py:**

from typing import List, Tuple

import random

import numpy as np

from paths import get\_path\_from\_cache\_or\_calculate, get\_path\_length\_from\_cache\_or\_calculate

from map import plot\_generation\_data

from settings import output\_gene\_crossover, output\_gene\_mutation, plot\_graph\_genetic

path\_cache = {}

path\_length\_cache = {}

# Счетчик проверок путей

path\_check\_count: int = 0

# Мутация - инвертирование части отрезка

def inverse\_mutation(route: List[int]) -> List[int]:

if output\_gene\_mutation:

print(f"route before inverse mutation: {route}")

mutated\_route = route.copy()

idx1, idx2 = sorted(random.sample(range(len(route)), 2))

mutated\_route[idx1:idx2 + 1] = reversed(mutated\_route[idx1:idx2 + 1])

if output\_gene\_mutation:

print(f"route after inverse mutation: {mutated\_route}")

return mutated\_route

# Мутация - перемешивание части отрезка

def scramble\_mutation(route: List[int]) -> List[int]:

if output\_gene\_mutation:

print(f"route before scramble mutation: {route}")

mutated\_route = route.copy()

idx1, idx2 = sorted(random.sample(range(len(route)), 2))

sub\_route = mutated\_route[idx1:idx2 + 1]

random.shuffle(sub\_route)

mutated\_route[idx1:idx2 + 1] = sub\_route

if output\_gene\_mutation:

print(f"route after scramble mutation: {mutated\_route}")

return mutated\_route

# Мутация - перестановка двух случайных элементов

def swap\_mutation(route: List[int]) -> List[int]:

if output\_gene\_mutation:

print(f"route before swap mutation: {route}")

mutated\_route = route.copy()

idx1, idx2 = random.sample(range(len(route)), 2)

mutated\_route[idx1], mutated\_route[idx2] = mutated\_route[idx2], mutated\_route[idx1]

if output\_gene\_mutation:

print(f"route after swap mutation: {mutated\_route}")

return mutated\_route

# Диверсификация мутаций

def diversified\_mutation(\_route: List[int], \_generation: int, \_max\_generations: int) -> List[int]:

mutation\_choice = random.random()

if \_generation < \_max\_generations \* 0.5: # Для начальных поколений более случайные мутации

if mutation\_choice < 0.33:

return inverse\_mutation(\_route)

elif mutation\_choice < 0.66:

return scramble\_mutation(\_route)

else:

return swap\_mutation(\_route)

else: # Для поздних поколений более "серьезные" мутации

if mutation\_choice < 0.5:

return inverse\_mutation(\_route)

else:

return scramble\_mutation(\_route)

# Partially Mapped Crossover (PMX)

def pmx\_crossover(parent1: List[int], parent2: List[int]) -> Tuple[List[int], List[int]]:

size = len(parent1)

start, end = sorted(random.sample(range(size), 2))

if output\_gene\_crossover:

print(f"parent1: {parent1}")

print(f"parent2: {parent2}")

# Создание двух детей

child1 = [-1] \* size

child2 = [-1] \* size

# Копирование части отрезка от родителя 1 в child1, и от родителя 2 в child2

for i in range(start, end + 1):

child1[i] = parent1[i]

child2[i] = parent2[i]

if output\_gene\_crossover:

print(f"child1 (after part copy): {child1}")

print(f"child2 (after part copy): {child2}")

# Заполнение оставшихся пустых мест в child1

for i in range(size):

if child1[i] == -1:

# Пропускаем элементы, которые уже есть в перекопированной части

current\_element = parent2[i]

while current\_element in child1: # Пропускаем элементы, которые уже есть в child1

idx = parent1.index(current\_element) # Ищем, где элемент находится в родителе 2

current\_element = parent2[idx] # Получаем элемент из второго родителя

child1[i] = current\_element

# Заполнение оставшихся пустых мест в child2

for i in range(size):

if child2[i] == -1:

current\_element = parent1[i]

while current\_element in child2: # Пропускаем элементы, которые уже есть в child2

idx = parent2.index(current\_element) # Ищем, где элемент находится в родителе 1

current\_element = parent1[idx] # Получаем элемент из первого родителя

child2[i] = current\_element

# Проверка, что оба ребенка корректно заполнились

if -1 in child1 or -1 in child2:

raise ValueError("Не все позиции заполнены корректно.")

if output\_gene\_crossover:

print(f"child1 (final): {child1}")

print(f"child2 (final): {child2}")

return child1, child2

# Турнирный отбор с агрессивным отбором лучших особей

def tournament\_selection(population: List[List[int]], \_terrain\_map: np.ndarray, \_tournament\_size: int,

points: List[Tuple[int, int]]) -> List[int]:

tournament = random.sample(population, \_tournament\_size)

evaluated\_tournament = []

for route in tournament:

total\_length = 0

for i in range(len(route) - 1):

# Подсчитываем количество проверок путей

global path\_check\_count

path\_check\_count += 1 # Каждая проверка пути увеличивает счетчик

total\_length += get\_path\_length\_from\_cache\_or\_calculate(points[route[i]], points[route[i + 1]],

\_terrain\_map,

path\_cache, path\_length\_cache)

evaluated\_tournament.append((route, total\_length))

evaluated\_tournament.sort(key=lambda x: x[1])

best\_route = evaluated\_tournament[0][0]

assert best\_route is not None, "Лучший маршрут не может быть пустым"

assert len(best\_route) == len(population[0]), f"Размер маршрута несоответствует ожиданиям."

assert check\_valid\_route(best\_route), "Лучший маршрут невалиден."

return best\_route

# Адаптивный размер турнира

def adaptive\_tournament\_selection(population: List[List[int]], \_terrain\_map: np.ndarray,

generation: int, \_generations: int, \_tournament\_base\_size: int,

points: List[Tuple[int, int]]) -> List[int]:

tournament\_size = max(\_tournament\_base\_size, int(\_tournament\_base\_size \* (1 + generation / \_generations)))

return tournament\_selection(population, \_terrain\_map, tournament\_size, points)

# Проверка маршрута

def check\_valid\_route(route: List[int]) -> bool:

if None in route or -1 in route or len(route) < 2:

return False

return len(set(route)) == len(route)

def genetic\_algorithm\_routing(\_start: Tuple[int, int], \_points: List[Tuple[int, int]], \_end: Tuple[int, int],

\_terrain\_map: np.ndarray, \_population\_size: int, \_generations: int,

\_mutation\_rate: float, \_tournament\_size: int) -> List[

Tuple[int, int]]:

# Убираем начальную и конечную точку из списка точек

points = [point for point in \_points if point != \_start and point != \_end]

# Создание начальной популяции маршрутов с индексами точек (исключая начальную и конечную)

population = [random.sample(range(len(points)), len(points)) for \_ in range(\_population\_size)]

best\_solution = None

best\_length = float('inf')

# Списки для хранения данных для графика

generation\_lengths = [] # Список длин путей для каждого поколения

average\_lengths = [] # Список средних длин путей для каждого поколения

min\_lengths = [] # Список минимальных длин путей для каждого поколения

for generation in range(\_generations):

evaluated\_population = []

generation\_total\_length = 0 # Для подсчета средней длины в поколении

generation\_min\_length = float('inf') # Для подсчета минимальной длины в поколении

for route in population:

# Вставляем начальную и конечную точку в начало и конец маршрута

full\_route = [\_start] + [points[i] for i in route] + [\_end]

final\_path = []

for j in range(len(full\_route) - 1):

segment = get\_path\_from\_cache\_or\_calculate(full\_route[j], full\_route[j + 1], \_terrain\_map, path\_cache,

path\_length\_cache)

final\_path.extend(segment[:])

length = 0.0

for i in range(len(final\_path) - 1):

length += get\_path\_length\_from\_cache\_or\_calculate(final\_path[i], final\_path[i + 1], \_terrain\_map,

path\_cache, path\_length\_cache)

evaluated\_population.append((route, length))

generation\_total\_length += length

# Обновляем минимальную длину для поколения

if length < generation\_min\_length:

generation\_min\_length = length

if length < best\_length:

best\_length = length

best\_solution = full\_route

# Сортируем популяцию по длине пути

evaluated\_population.sort(key=lambda x: x[1])

best\_routes = [route for route, \_ in evaluated\_population[:\_population\_size // 2]]

# Сохраняем информацию о длинах для графика

generation\_lengths.append([length for \_, length in evaluated\_population])

average\_lengths.append(generation\_total\_length / len(population))

min\_lengths.append(generation\_min\_length)

# Создаем новую популяцию

new\_population = best\_routes[:]

# Создание новой популяции с двумя детьми на каждой итерации

while len(new\_population) < \_population\_size:

parent1 = adaptive\_tournament\_selection(best\_routes, \_terrain\_map, generation, \_generations,

\_tournament\_size, points)

parent2 = adaptive\_tournament\_selection(best\_routes, \_terrain\_map, generation, \_generations,

\_tournament\_size, points)

# Получаем два ребенка от кроссовера

child1, child2 = pmx\_crossover(parent1, parent2)

# Применяем мутацию

if random.random() < \_mutation\_rate:

child1 = diversified\_mutation(child1, generation, \_generations)

child2 = diversified\_mutation(child2, generation, \_generations)

# Добавляем детей в новую популяцию, если они корректны

if check\_valid\_route(child1):

new\_population.append(child1)

if check\_valid\_route(child2):

new\_population.append(child2)

population = new\_population

if generation % 10 == 0:

print(f"Поколение {generation}, длина лучшего маршрута: {best\_length}")

if best\_solution is None:

raise ValueError("Не найдено корректного маршрута.")

final\_path = []

for j in range(len(best\_solution) - 1):

segment = get\_path\_from\_cache\_or\_calculate(best\_solution[j], best\_solution[j + 1], \_terrain\_map, path\_cache,

path\_length\_cache)

final\_path.extend(segment[1:])

final\_path = [\_start] + final\_path # Добавляем начальную точку в начало маршрута

print(f"Количество проверок путей: {path\_check\_count}")

print(f"Количество реальных вычислений путей: {len(path\_length\_cache)}")

path\_cache.clear()

path\_length\_cache.clear()

# Построение графика, если plot\_graph == True

if plot\_graph\_genetic:

plot\_generation\_data(generation\_lengths, average\_lengths, min\_lengths, \_generations)

return final\_path

**main.py:**

from map import generate\_map, prepare\_basic\_map, generate\_random\_points, visualize\_route

from neighbor import nearest\_neighbor\_routing

from genetic import genetic\_algorithm\_routing

from brute\_force import brute\_force\_routing

from annealing import simulated\_annealing\_routing

from paths import a\_star\_path, calculate\_path\_length

import numpy as np

import matplotlib.pyplot as plt

from typing import List

from settings import \*

# Задание точек маршрута

points: List[Tuple[int, int]]

if choose\_points == 0:

points = generate\_random\_points((0, 0), (512, 512), num\_random\_points)

elif choose\_points == 8:

points = [(33, 273), (174, 173), (269, 12), (386, 477), (222, 454), (245, 152), (42, 337), (354, 279)]

elif choose\_points == 10:

points = [(300, 157), (72, 36), (367, 366), (42, 337), (354, 279), (489, 307), (476, 344),

(224, 231), (157, 187), (369, 457)]

elif choose\_points == 20:

points = [(139, 113), (195, 252), (21, 494), (45, 407), (43, 280), (49, 340), (338, 31),

(353, 149), (173, 13), (271, 139), (23, 274), (218, 395), (490, 110), (369, 379),

(367, 401), (79, 250), (214, 109), (245, 412), (191, 311), (214, 443)]

else:

points = generate\_random\_points((0, 0), (512, 512), num\_random\_points)

# Выводим результат

print("Точки маршрута:", points)

# Генерация карты

terrain\_map: np.ndarray = generate\_map(width, height, scale, octaves, persistence, lacunarity)

if do\_original:

# Визуализация пути

prepare\_basic\_map(start, end, points, terrain\_map)

plt.title("Оригинальный ландшафт")

plt.legend()

plt.show()

if do\_a\_star\_forward:

# Поиск маршрута от начала до конца

path\_a\_star\_forward: List[Tuple[int, int]] = a\_star\_path(start, end, terrain\_map, show\_visited\_nodes)

path\_length = round(calculate\_path\_length(path\_a\_star\_forward, terrain\_map), 5)

visualize\_route(path\_a\_star\_forward, path\_length, "A\* start - end", start, [], end, terrain\_map)

if do\_a\_star\_backward:

# Поиск маршрута от конца до начала

path\_a\_star\_backward: List[Tuple[int, int]] = a\_star\_path(end, start, terrain\_map, show\_visited\_nodes)

path\_length = round(calculate\_path\_length(path\_a\_star\_backward, terrain\_map), 5)

visualize\_route(path\_a\_star\_backward, path\_length, "A\* end - start", end, [], start, terrain\_map)

if do\_brute\_force:

# Поиск маршрута методом грубой силы

path\_brute\_force: List[Tuple[int, int]] = brute\_force\_routing(start, points, end, terrain\_map)

path\_length = round(calculate\_path\_length(path\_brute\_force, terrain\_map), 5)

visualize\_route(path\_brute\_force, path\_length, "Алгоритм грубой силы", start, points, end, terrain\_map)

if do\_nearest\_neighbor:

# Поиск маршрута методом ближайшего соседа

path\_nearest\_neighbor: List[Tuple[int, int]] = nearest\_neighbor\_routing(start, points, end, terrain\_map)

path\_length = round(calculate\_path\_length(path\_nearest\_neighbor, terrain\_map), 5)

visualize\_route(path\_nearest\_neighbor, path\_length, "Алгоритм ближайшего соседа", start, points, end, terrain\_map)

if do\_annealing:

# Поиск маршрута методом симулированного отжига

path\_annealing: List[Tuple[int, int]] = simulated\_annealing\_routing(start, points, end, terrain\_map, initial\_temp,

cooling\_rate, iterations)

path\_length = round(calculate\_path\_length(path\_annealing, terrain\_map), 5)

visualize\_route(path\_annealing, path\_length, "Алгоритм симулированного отжига", start, points, end, terrain\_map)

if do\_genetic:

# Генетический поиск маршрута

path\_genetic: List[Tuple[int, int]] = genetic\_algorithm\_routing(start, points, end, terrain\_map,

population\_size, generations,

mutation\_rate, tournament\_size)

path\_length = round(calculate\_path\_length(path\_genetic, terrain\_map), 5)

visualize\_route(path\_genetic, path\_length, "Генетический алгоритм", start, points, end, terrain\_map)

**map.py:**

import noise

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.patches import FancyArrowPatch

from typing import List, Tuple

import random

# Генерация карты

def generate\_map(\_width: int, \_height: int, \_scale: float, \_octaves: int, \_persistence: float,

\_lacunarity: float) -> np.ndarray:

map\_data: np.ndarray = np.zeros((\_height, \_width))

for y in range(\_height):

for x in range(\_width):

# Генерация значения шума Перлина в диапазоне [-1, 1]

raw\_value = noise.snoise2(

x / \_scale,

y / \_scale,

octaves=\_octaves,

persistence=\_persistence,

lacunarity=\_lacunarity,

repeatx=1024,

repeaty=1024,

base=42

)

# Нормализация в диапазон [0, 1]

normalized\_value = (raw\_value + 1) / 2 # Переводим в диапазон [0, 1]

# Записываем в карту

map\_data[y][x] = normalized\_value

return map\_data

def prepare\_basic\_map(\_start: Tuple[int, int], \_end: Tuple[int, int], \_points: List[Tuple[int, int]],

\_terrain\_map: np.ndarray) -> None:

plt.figure(figsize=(10, 10))

plt.imshow(\_terrain\_map, cmap='terrain')

plt.colorbar(label="Высота")

# Точки маршрута (пурпурные, большие, с кругами)

if len(\_points) != 0:

plt.scatter([p[1] for p in \_points], [p[0] for p in \_points], color="purple", s=100,

marker='o', label="Точки маршрута", zorder=3, alpha=0.7, edgecolor='black')

plot\_start\_and\_end(\_start, \_end)

def plot\_start\_and\_end(\_start: Tuple[int, int], \_end: Tuple[int, int]) -> None:

# Начальная точка (зеленая, большая, с треугольником)

plt.scatter(\_start[1], \_start[0], color="green", s=150, marker='^', label="Начальная точка", zorder=3,

edgecolor='black', linewidth=2)

# Конечная точка (красная, большая, с квадратом)

plt.scatter(\_end[1], \_end[0], color="red", s=150, marker='s', label="Конечная точка", zorder=3,

edgecolor='black', linewidth=2)

def draw\_path(\_path: List[Tuple[int, int]]) -> None:

for i in range(len(\_path) - 1):

# Рисуем линии между точками

plt.plot([\_path[i][1], \_path[i + 1][1]], [\_path[i][0], \_path[i + 1][0]], color="red", linewidth=1, zorder=1)

if i % 30 == 0: # Рисуем стрелки каждые 30 шагов

# Используем FancyArrowPatch для более плавных стрелок

arrow = FancyArrowPatch(

(\_path[i][1], \_path[i][0]), # Начало стрелки

(\_path[i + 1][1], \_path[i + 1][0]), # Направление стрелки

mutation\_scale=10, # Масштаб стрелки

color="blue", # Цвет стрелки

arrowstyle="->", # Стиль стрелки

lw=2, # Толщина стрелки

alpha=0.7, # Прозрачность

zorder=2 # Порядок наложения

)

plt.gca().add\_patch(arrow)

def visualize\_route(\_path: List[Tuple[int, int]], \_path\_length: float, \_algorithm\_name: str, \_start: Tuple[int, int],

\_points: List[Tuple[int, int]], \_end: Tuple[int, int], \_terrain\_map: np.ndarray):

# Расчет длины пути и округление

print(f"Длина пути ({\_algorithm\_name}): {\_path\_length}\n")

# Визуализация пути

prepare\_basic\_map(\_start, \_end, \_points, \_terrain\_map)

draw\_path(\_path)

# Заголовок с длиной пути

plt.title(f"{\_algorithm\_name}, Длина пути: {\_path\_length}")

plt.legend()

plt.show()

# Функция для генерации случайных точек в пределах определенной области

def generate\_random\_points(\_start: Tuple[int, int], \_end: Tuple[int, int], \_num\_points: int, margin: int = 10) -> List[

Tuple[int, int]]:

\_points: List[Tuple[int, int]] = []

# Определяем границы, чтобы точки не были слишком близко к краям

x\_min = min(\_start[0], \_end[0]) + margin

x\_max = max(\_start[0], \_end[0]) - margin

y\_min = min(\_start[1], \_end[1]) + margin

y\_max = max(\_start[1], \_end[1]) - margin

while len(\_points) < \_num\_points:

# Генерируем случайные точки в пределах границ

new\_point = (random.randint(x\_min, x\_max), random.randint(y\_min, y\_max))

# Проверяем, что новая точка не слишком близка к уже существующим точкам

if not any(abs(new\_point[0] - p[0]) < 10 and abs(new\_point[1] - p[1]) < 10 for p in \_points):

\_points.append(new\_point)

return \_points

def plot\_generation\_data(\_generation\_lengths, \_average\_lengths, \_min\_lengths,

\_generations: int):

plt.figure(figsize=(10, 6))

# Отображаем облако значений длин путей для каждой особи

for generation in range(\_generations):

plt.scatter([generation] \* len(\_generation\_lengths[generation]), \_generation\_lengths[generation],

color='grey', alpha=0.3)

# Отображаем среднюю длину пути для каждого поколения

plt.plot(range(\_generations), \_average\_lengths, color='red', label='Средняя длина пути', linewidth=3)

# Отображаем минимальную длину пути для каждого поколения

plt.plot(range(\_generations), \_min\_lengths, color='green', label='Минимальная длина пути', linewidth=3)

plt.title("Зависимость длин путей от поколения")

plt.xlabel("Поколение")

plt.ylabel("Длина пути")

plt.legend()

plt.grid(True)

plt.show()

def plot\_annealing\_data(\_costs, \_iterations: int):

plt.figure(figsize=(10, 6))

# Отображаем длину пути для каждой итерации

plt.plot(range(\_iterations), \_costs, color='blue', label='Длина пути', linewidth=2)

plt.title("Зависимость длины пути от итерации в алгоритме имитации отжига")

plt.xlabel("Итерация")

plt.ylabel("Длина пути")

plt.legend()

plt.grid(True)

plt.show()

def plot\_visited\_nodes(\_start: Tuple[int, int], \_end: Tuple[int, int], \_visited\_nodes: set[Tuple[int, int]],

\_terrain\_map: np.ndarray) -> None:

# Визуализируем карту

plt.figure(figsize=(10, 10))

plt.imshow(\_terrain\_map, cmap='terrain')

plt.colorbar(label="Высота")

# Отображаем все посещенные узлы

visited\_x = [pos[1] for pos in \_visited\_nodes]

visited\_y = [pos[0] for pos in \_visited\_nodes]

plt.scatter(visited\_x, visited\_y, color="red", s=10, label="Посещенные точки", zorder=3)

plot\_start\_and\_end(\_start, \_end)

plt.legend()

plt.show() # Показываем картину

**neighbor.py:**

from paths import get\_path\_from\_cache\_or\_calculate, get\_path\_length\_from\_cache\_or\_calculate

from typing import List, Tuple

import numpy as np

path\_cache = {}

path\_length\_cache = {}

# Счетчики для подсчета проверок

path\_check\_count = 0 # Для подсчета проверок путей

# Функция маршрутизации методом ближайшего соседа

def nearest\_neighbor\_routing(\_start: Tuple[int, int], \_points: List[Tuple[int, int]], \_end: Tuple[int, int],

\_terrain\_map: np.ndarray) -> List[Tuple[int, int]]:

global path\_check\_count

unvisited\_points = \_points.copy()

current\_point = \_start

final\_path = [\_start]

while unvisited\_points:

# Находим ближайшую точку с учетом кэширования длины пути

closest\_point = min(unvisited\_points, key=lambda p: get\_path\_length\_from\_cache\_or\_calculate(

current\_point, p, \_terrain\_map, path\_cache, path\_length\_cache))

path\_check\_count += len(unvisited\_points)

# Получаем путь между текущей точкой и ближайшей с учётом кэширования

path\_segment = get\_path\_from\_cache\_or\_calculate(current\_point, closest\_point, \_terrain\_map, path\_cache,

path\_length\_cache)

# Увеличиваем счетчик проверок путей

path\_check\_count += 1

final\_path.extend(path\_segment[1:]) # Добавляем путь без начальной точки сегмента

# Обновляем текущую точку и убираем её из не посещённых

current\_point = closest\_point

unvisited\_points.remove(closest\_point)

# Добавляем путь к конечной точке с учетом кэширования

final\_path\_segment = get\_path\_from\_cache\_or\_calculate(current\_point, \_end, \_terrain\_map, path\_cache,

path\_length\_cache)

# Увеличиваем счетчик проверок пути и длины пути

path\_check\_count += 1

final\_path.extend(final\_path\_segment[1:])

# Выводим количество проверок

print(f"Количество проверок путей: {path\_check\_count}")

print(f"Количество реальных вычислений путей: {len(path\_length\_cache)}")

path\_cache.clear() # Очищаем кэш путей после завершения маршрута

path\_length\_cache.clear() # Очищаем кэш длин путей после завершения маршрута

return final\_path

**paths.py:**

from typing import List, Tuple, Dict

from map import plot\_visited\_nodes

import heapq

import numpy as np

height\_weight: float = 200

# Направления для поиска соседей

directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

# Функция для вычисления длины маршрута с учетом ландшафта

def calculate\_path\_length(path: List[Tuple[int, int]], \_terrain\_map: np.ndarray) -> float:

length = 0

for j in range(len(path) - 1):

x1, y1 = path[j]

x2, y2 = path[j + 1]

terrain\_cost = calculate\_height((x1, y1), (x2, y2), \_terrain\_map)

length += heuristic((x1, y1), (x2, y2)) + terrain\_cost

return length

# Получение пути из кэша (с добавлением кэша длины пути)

def get\_path\_from\_cache\_or\_calculate(start: Tuple[int, int], end: Tuple[int, int], terrain\_map: np.ndarray,

\_path\_cache: dict, \_path\_length\_cache: dict) -> List[Tuple[int, int]]:

# Проверка, есть ли уже путь в кэше

if (start, end) in \_path\_cache:

return \_path\_cache[(start, end)]

# Если пути нет в кэше, вычисляем его с использованием A\* и сохраняем в кэш

path = a\_star\_path(start, end, terrain\_map)

\_path\_cache[(start, end)] = path

# Также вычисляем длину пути и сохраняем ее в кэш

path\_length = calculate\_path\_length(path, terrain\_map)

\_path\_length\_cache[(start, end)] = path\_length

return path

# Получение длины пути из кэша (если есть)

def get\_path\_length\_from\_cache\_or\_calculate(start: Tuple[int, int], end: Tuple[int, int], terrain\_map: np.ndarray,

\_path\_cache: dict, \_path\_length\_cache: dict) -> float:

# Проверка, есть ли длина пути в кэше

if (start, end) in \_path\_length\_cache:

return \_path\_length\_cache[(start, end)]

# Если длина пути нет в кэше, получаем путь из кэша или вычисляем его

path = get\_path\_from\_cache\_or\_calculate(start, end, terrain\_map, \_path\_cache, \_path\_length\_cache)

# Рассчитываем и сохраняем длину пути в кэш

path\_length = calculate\_path\_length(path, terrain\_map)

\_path\_length\_cache[(start, end)] = path\_length

return path\_length

# Класс для узлов в графе

class Node:

def \_\_init\_\_(self, position: Tuple[int, int], g\_cost: float = 0.0, h\_cost: float = 0.0,

parent: 'Node' = None) -> None:

self.position: Tuple[int, int] = position

self.g\_cost: float = g\_cost # Стоимость пути от старта до текущего узла

self.h\_cost: float = h\_cost # Эвристическая стоимость от текущего узла до цели

self.f\_cost: float = g\_cost + h\_cost # Общая стоимость (g + h)

self.parent: 'Node' = parent

def \_\_lt\_\_(self, other: 'Node') -> bool:

return self.f\_cost < other.f\_cost

def \_\_eq\_\_(self, other: 'Node') -> bool:

return self.position == other.position

# Эвристическая функция (расстояние Евклида)

def heuristic(a: Tuple[int, int], b: Tuple[int, int]) -> float:

# return np.sqrt((a[0] - b[0]) \*\* 2 + (a[1] - b[1]) \*\* 2 + calculate\_height(a, b, \_terrain\_map) \*\* 2)

return abs(a[0] - b[0]) + abs(a[1] - b[1])

# Функция для вычисления высоты между точками

def calculate\_height(\_from: Tuple[int, int], \_to: Tuple[int, int], \_terrain\_map: np.ndarray) -> float:

height1 = float(\_terrain\_map[\_from[0], \_from[1]])

height2 = float(\_terrain\_map[\_to[0], \_to[1]])

return height\_weight \* abs(height2 - height1)

# Основная функция поиска пути с использованием A\*

def a\_star\_path(start: Tuple[int, int], end: Tuple[int, int], \_terrain\_map: np.ndarray,

\_show\_visited\_nodes: bool = False) -> List[Tuple[int, int]]:

open\_list: List[Node] = [] # Очередь с приоритетами

closed\_set: set[Tuple[int, int]] = set() # Множество посещенных узлов

came\_from: Dict[Tuple[int, int], Node] = {} # Для восстановления пути

g\_costs: Dict[Tuple[int, int], float] = {} # Стоимости пути для каждого узла

start\_node: Node = Node(start, 0, heuristic(start, end))

heapq.heappush(open\_list, start\_node)

g\_costs[start] = 0 # Начальная стоимость пути для старта

path = [] # Список для хранения пути

while open\_list:

current\_node: Node = heapq.heappop(open\_list)

# Если нашли путь до цели

if current\_node.position == end:

# Восстанавливаем путь от конечного узла до начального

while current\_node:

path.append(current\_node.position)

current\_node = came\_from.get(current\_node.position)

if \_show\_visited\_nodes:

plot\_visited\_nodes(start, end, closed\_set, \_terrain\_map)

return path[::-1]

closed\_set.add(current\_node.position)

for neighbor in directions:

neighbor\_pos = (current\_node.position[0] + neighbor[0], current\_node.position[1] + neighbor[1])

# Проверка, что сосед в пределах карты

if 0 <= neighbor\_pos[0] < \_terrain\_map.shape[0] and 0 <= neighbor\_pos[1] < \_terrain\_map.shape[1]:

if neighbor\_pos in closed\_set:

continue

# Вычисление новых g и h стоимостей для соседа

g\_cost = current\_node.g\_cost + calculate\_height(current\_node.position, neighbor\_pos, \_terrain\_map)

h\_cost = heuristic(neighbor\_pos, end)

# Если сосед уже в открытом списке с меньшей или равной стоимостью, пропускаем его

if neighbor\_pos in g\_costs and g\_costs[neighbor\_pos] <= g\_cost:

continue

g\_costs[neighbor\_pos] = g\_cost

neighbor\_node = Node(neighbor\_pos, g\_cost, h\_cost, current\_node)

heapq.heappush(open\_list, neighbor\_node)

came\_from[neighbor\_pos] = current\_node # Отслеживаем путь

return []

**settings.py:**

from typing import Tuple

# Параметры выполняемых тестовых опытов

do\_original: bool = False

show\_visited\_nodes: bool = True

do\_a\_star\_forward: bool = False

do\_a\_star\_backward: bool = False

# Параметры выполняемых опытов

do\_brute\_force: bool = False

do\_nearest\_neighbor: bool = True

do\_annealing: bool = True

do\_genetic: bool = True

# Параметры карты

width: int = 512

height: int = 512

scale: float = 100.0 # Масштаб шума

# Параметры шума

octaves: int = 10 # Количество октав

persistence: float = 0.5 # Влияние каждой октавы

lacunarity: float = 2.0 # Частота осцилляций

# Параметры маршрута

start: Tuple[int, int] = (100, 100) # Начальная точка

end: Tuple[int, int] = (400, 400) # Конечная точка

num\_random\_points: int = 0 # Количество случайных точек

choose\_points: int = 20 # 0 Генерировать ли случайные числа, 8 чисел, 10 чисел, 20 чисел

# Параметры генетического метода

population\_size: int = 100 # Размер популяции

generations: int = 100 # Количество поколений

mutation\_rate: float = 0.3 # Вероятность мутации

tournament\_size: int = population\_size // 10 # Количество агентов для отбора

output\_gene\_crossover: bool = False

output\_gene\_mutation: bool = False

plot\_graph\_genetic: bool = True

# Параметры симулированного отжига

initial\_temp: float = 2000

cooling\_rate: float = 0.995

iterations: int = 1000

plot\_graph\_annealing: bool = True

**ПЛАН-ГРАФИК**

**выполнения курсового проекта**

обучающегося Нечаев Б.П.

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование этапа работ | Трудоемкость выполнения, час. | Процент к общей трудоемкости выполнения | Срок предъявления консультанту |
| Получение и согласование задания | 0,5 | 0,5 | 6 неделя |
| Знакомство с литературой по теме курсовой работы | 2,5 | 7,5 | 7 неделя |
| Отрисовка карты с использованием шумов и визуализация графа | 5 | 15 | 9 неделя |
| Реализация алгоритма А\* | 5 | 16 | 9 неделя |
| Реализация метода грубой силы (Brute Force) | 2 | 5 | 9 неделя |
| Реализация метода ближайшего соседа | 2,5 | 10 | 10 неделя |
| Реализация алгоритма имитации отжига | 6 | 15 | 12 неделя |
| Реализация генетического алгоритма | 7 | 15 | 13 неделя |
| Исправление и улучшение кода | 2,5 | 8 | 14 неделя |
| Составление и оформление пояснительной записки и подготовка к защите | 2,5 | 7,5 | 15 неделя |
| Защита | 0,5 | 0,5 | 16 неделя |
| Итого | 36 | 100 |  |