**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](#_Toc185014353)

[1. Теоретическая Часть 6](#_Toc185014354)

[1.1. Математическая постановка задачи 6](#_Toc185014355)

[1.2. Алгоритм А\*(A-star) 7](#_Toc185014356)

[1.3. Алгоритм ближайшего соседа 8](#_Toc185014357)

[1.4. Алгоритм имитации отжига 9](#_Toc185014358)

[1.5. Генетический алгоритм 11](#_Toc185014359)

[2. Практическая Часть 15](#_Toc185014360)

[2.1. Общий вид программы 15](#_Toc185014361)

[2.2. Алгоритм А\* 16](#_Toc185014362)

[2.3. Алгоритм грубой силы 19](#_Toc185014363)

[2.4. Алгоритм ближайшего соседа 20](#_Toc185014364)

[2.5. Алгоритм имитации отжига 22](#_Toc185014365)

[2.6. Генетический алгоритм 22](#_Toc185014366)

[2.7. Пример работы программы 22](#_Toc185014367)

[Заключение 23](#_Toc185014368)

[Список литературы 24](#_Toc185014369)

[Приложение А 25](#_Toc185014370)

**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.

**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. 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.

**Математическая формулировка задачи**:

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

**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 алгоритм поиска кратчайшего пути, который находит оптимальный маршрут от начальной вершины к целевой в графе. Он сочетает в себе элементы поиска в ширину и жадного поиска, эффективно используя эвристическую функцию для ускорения поиска.

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. 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. 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.

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). Один из популярных методов скрещивания — это PMX (Partially Mapped Crossover), при котором наследуется часть маршрута от обоих родителей.

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**

* Гибкость: генетический алгоритм подходит для решения широкого круга задач оптимизации, в том числе и для задач с нелинейными зависимостями, сложными ограничениями и многими переменными. В случае VRP он может эффективно работать с различными видами затрат и ограничений, такими как ограничения по времени, грузоподъемности или географическим условиям.
* 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. 1.а), используя для этого шум Перлина из python библиотеки noise.

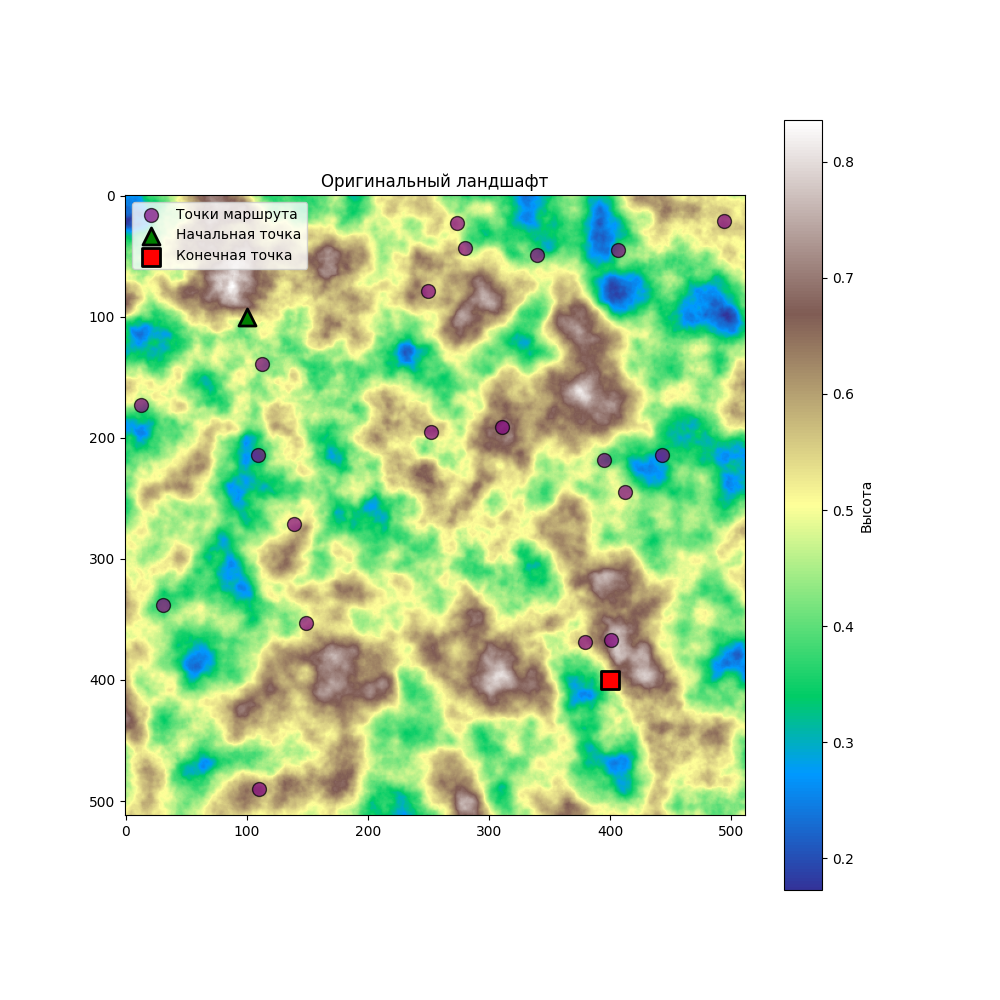
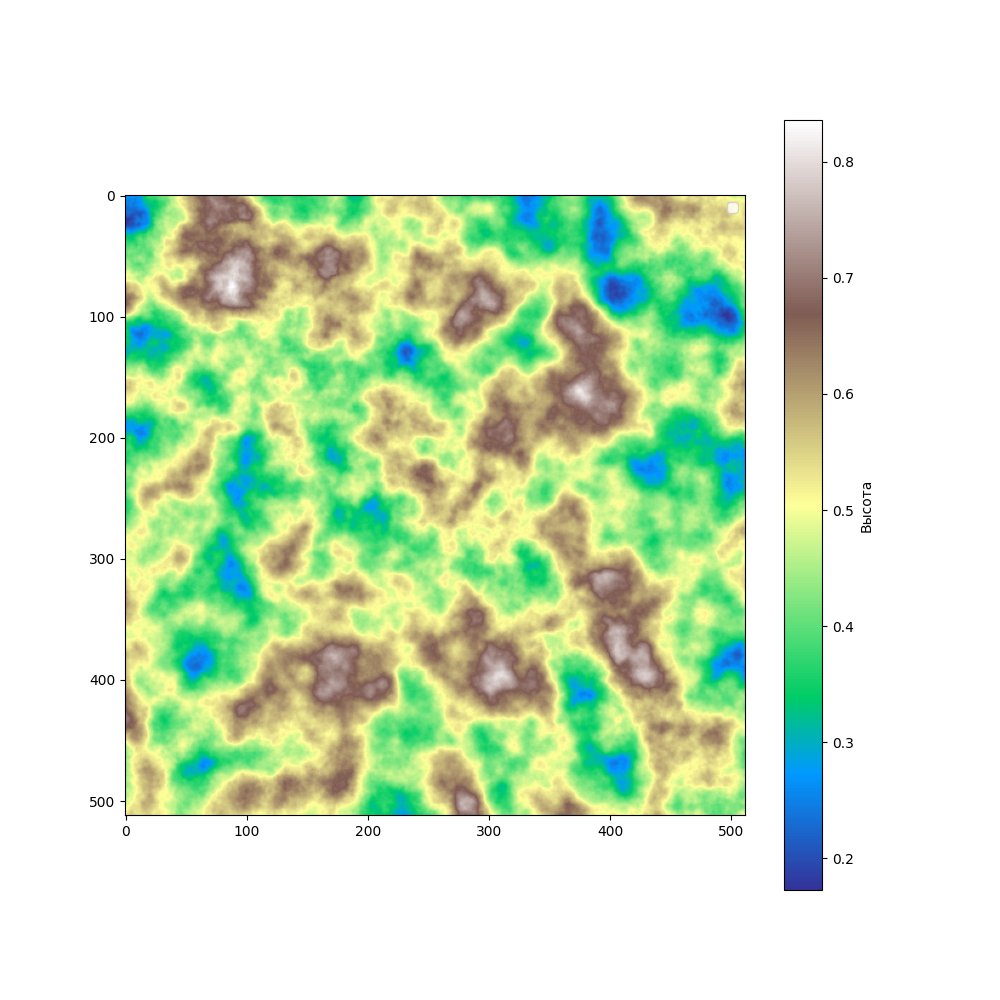


Рисунок 1. а) Сгенерированная карта высот

б) Ландшафт с расставленными точками

* Расстановка случайных или записанных точек начала, конца и некоторого числа промежуточных точек (рис. 1.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. 2). 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.

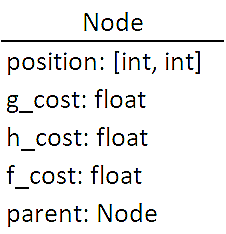


Рисунок 2. Класс Node

* 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. 3).

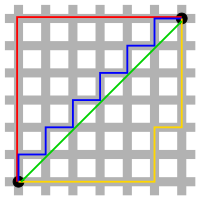


Рисунок 3. Манхэттенское расстояние

* 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. 4). Программа также вычисляет и выводит длину пути с учётом рельефа (включая изменения высот) и эвристической стоимости.

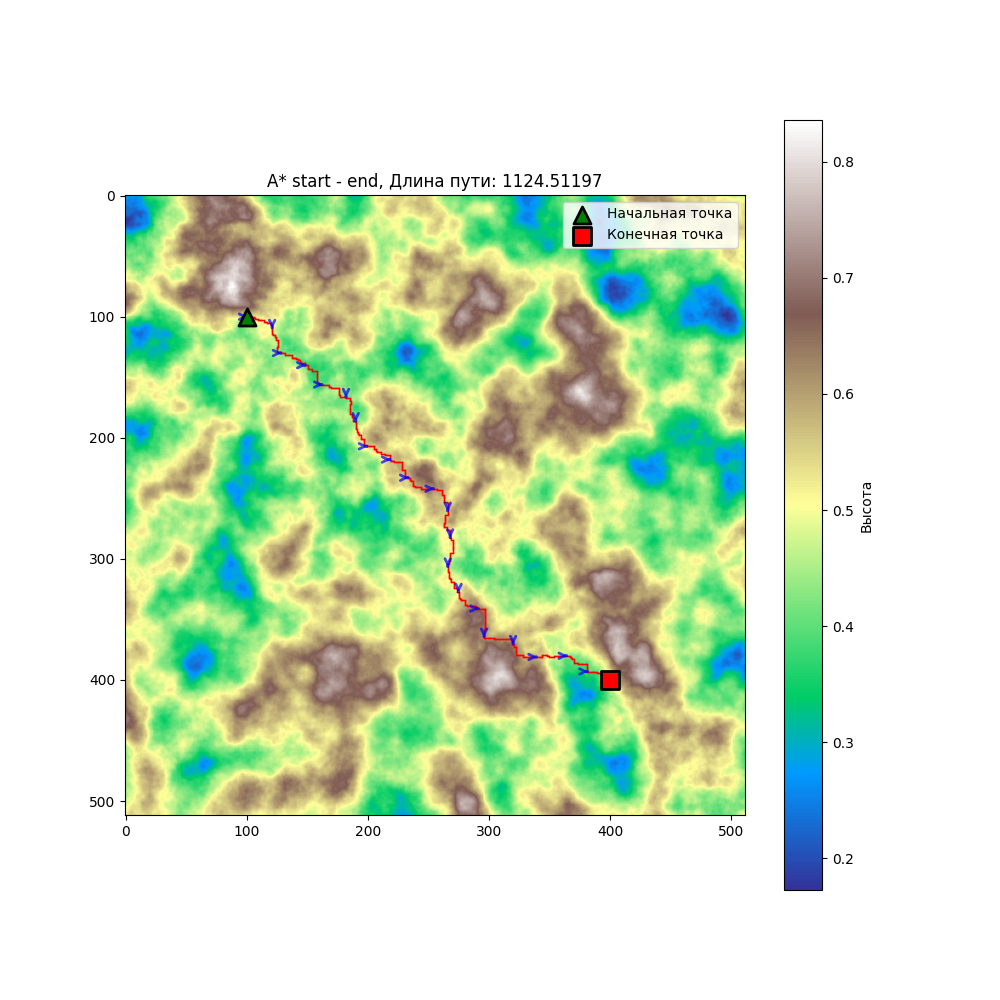


Рисунок 4. Рассчитанный путь А\* между двумя точками

Если включена опция отображения посещённых точек, программа выводит на экран все узлы, которые были обработаны на пути поиска (рис.5).

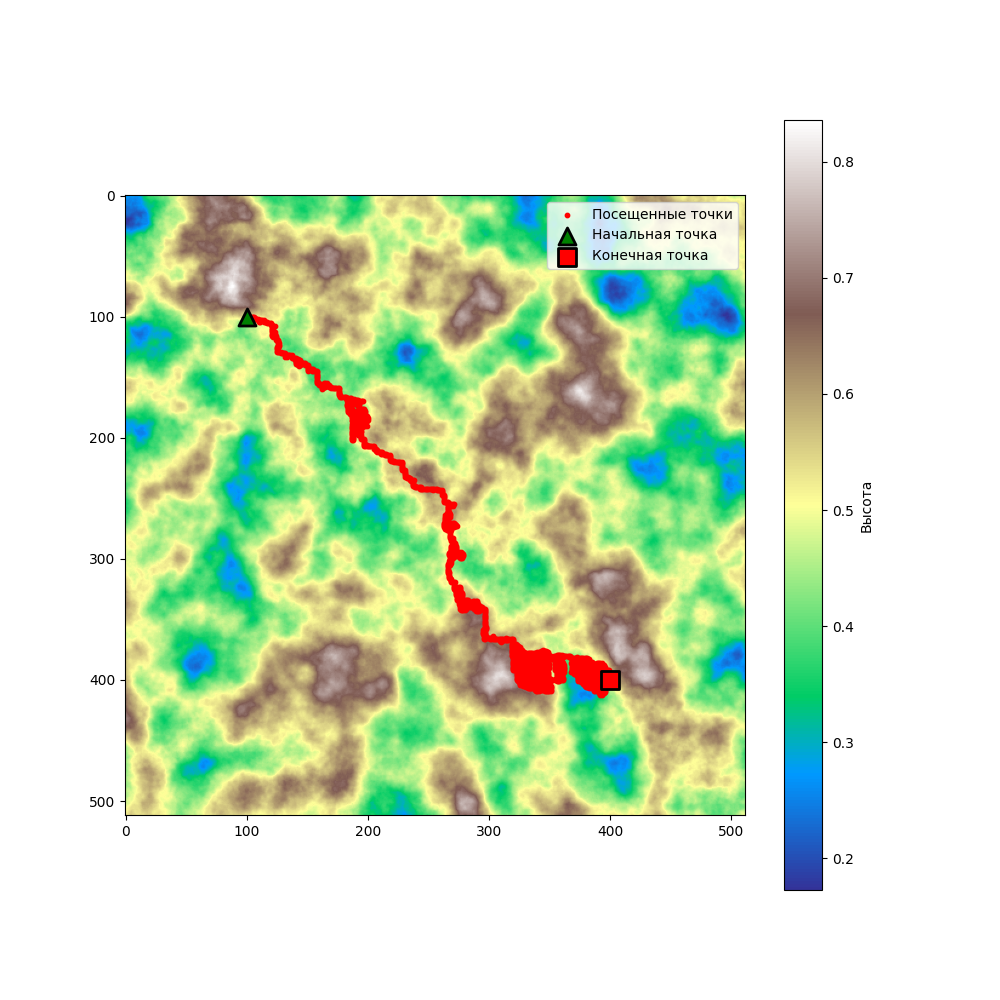


Рисунок 5. Посещённые во время работы алгоритма точки

## 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. 6).

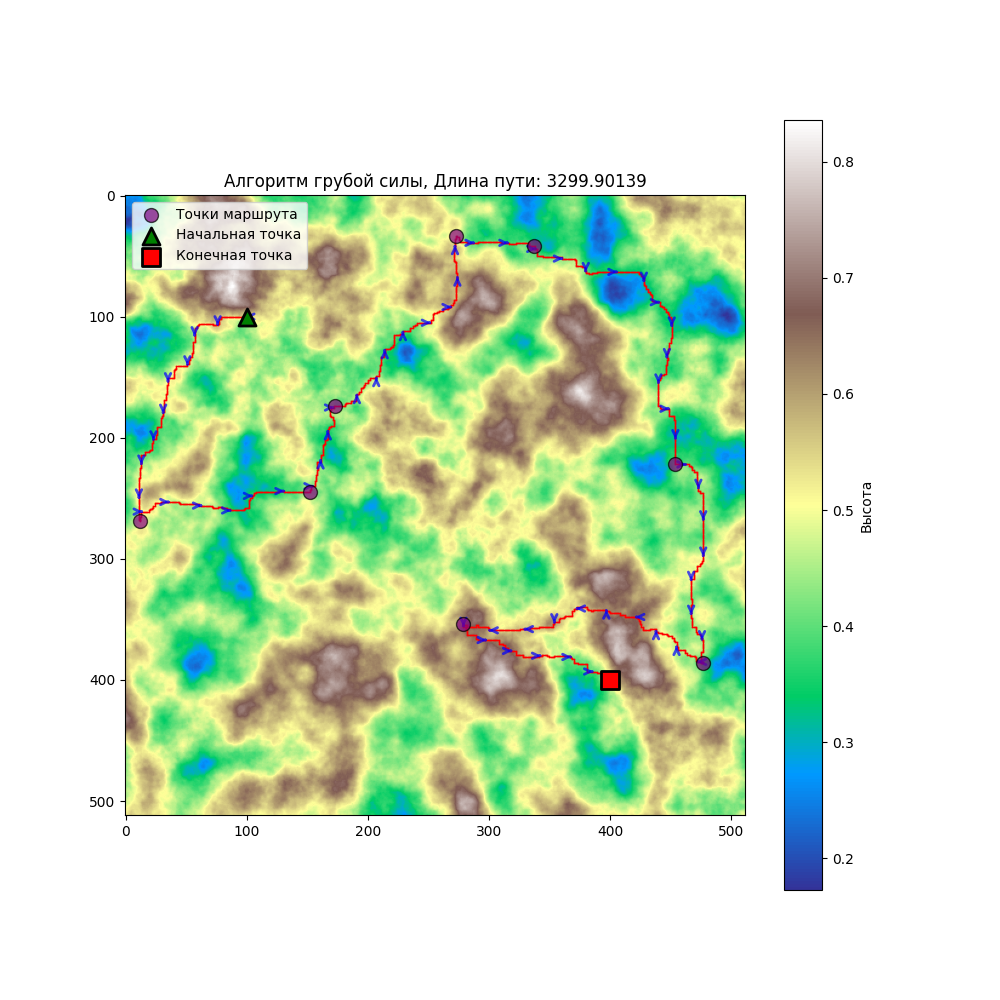


Рисунок 6. Путь, просчитанный алгоритмом грубой силы для 8 промежуточных точек

**Подсчёт затраченных 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. 7).

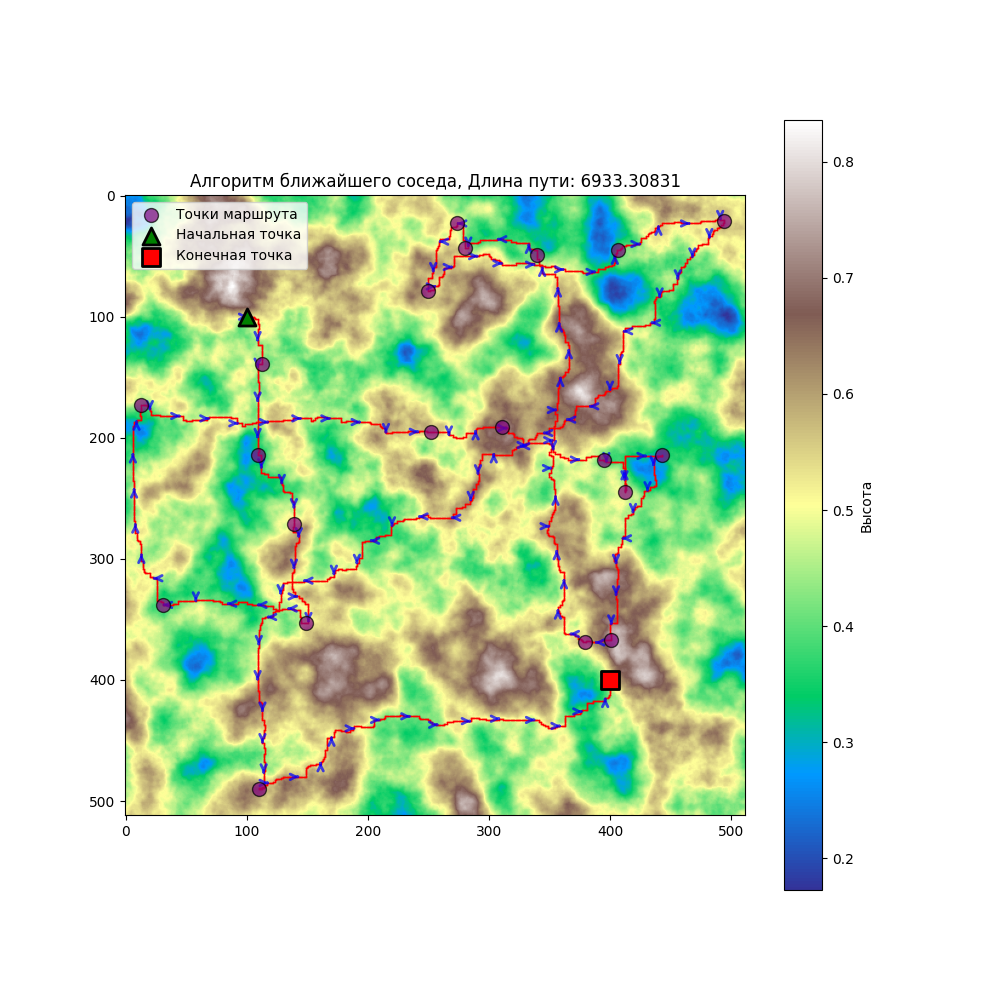
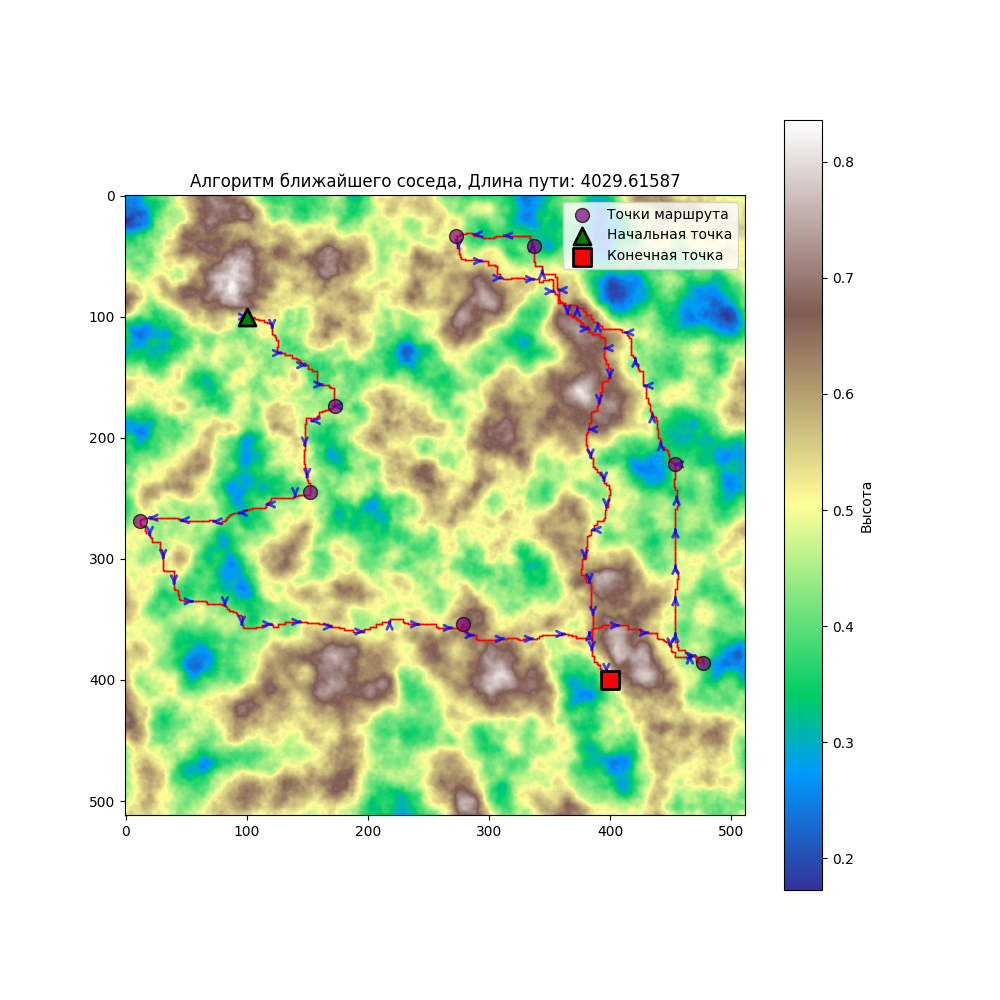


Рисунок 7. Путь, просчитанный алгоритмом ближайшего соседа для, а) 8 промежуточных точек, б) 20 промежуточных точек

**Подсчёт затраченных 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

Если сравнивать алгоритм ближайшего соседа с методом грубой силы, то можно отметить, хотя ближайший сосед часто находит более длинный путь, 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 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.

## Генетический алгоритм

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.

## Пример работы программы

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.

**Заключение**

В ходе проведенной работы были изучены алгоритма поиска максимального потока в сети, при помощи средства языка C++ этот алгоритм были реализованы.

Все задачи, поставленные в начале, были решены – реализована загрузка графа из файла. Был реализован алгоритм поиска максимального потока в сети Форда - Фалкерсона. Был визуализирован процесс поиска максимального поток. Было обеспечено взаимодействие пользователя с программой для просмотра каждого этапа работы алгоритма.

Хотя программа предоставляет эффективный способ вычисления и визуализации максимального потока в сети, есть возможности для улучшения её производительности и эффективности.

Одно из возможных улучшений - более эффективное хранение потоков, особенно для более разреженных графов. В настоящей реализации используется трехмерный массив для хранения потоков по каждому ребру на каждом шаге алгоритма. Однако для разреженных графов это может привести к избыточному расходу памяти. Более оптимальным подходом может быть использование разреженных структур данных, таких как списки смежности или разреженные матрицы, для эффективного хранения потоков только по рёбрам, через которые проходит поток. Это позволило бы значительно снизить объём используемой памяти и улучшить производительность программы для больших и разреженных графов.

**Список литературы**

1. 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.
2. Метод отжига — https://algorithmica.org/ru/annealing
3. Обход графа: поиск в глубину и поиск в ширину простыми словами на примере JavaScript — 2020. — https://habr.com/ru/articles/504374/

**Приложение А**

**Листинг программы**

#include <SFML/Graphics.hpp>  
#include <fstream>  
#include <iostream>  
#include <limits>  
#include <queue>  
#include <sstream>  
#include <vector>  
  
const sf::Color ColorBackground = sf::Color(255, 245, 238, 255);  
const sf::Color ColorNodeDefault = sf::Color(255, 250, 240, 255);  
const sf::Color ColorNodeSource = sf::Color(95, 187, 78, 255);  
const sf::Color ColorNodeSink = sf::Color(239, 113, 53, 255);  
const sf::Color ColorEdgeDefault = sf::Color(0, 0, 0, 255);  
const sf::Color ColorEdgeMore = sf::Color(95, 187, 78, 255);  
const sf::Color ColorEdgeLess = sf::Color(255, 0, 0, 255);  
const sf::Color ColorEdgeSame = sf::Color(230, 148, 0, 255);  
const sf::Color ColorTextFill = sf::Color(255, 245, 238, 255);  
const sf::Color ColorTextOutline = sf::Color(0, 0, 0, 255);  
  
struct Node {  
  size\_t id;  
  size\_t x, y;  
};  
  
struct Edge {  
  size\_t from, to;  
  size\_t capacity;  
};  
  
class Graph {  
private:  
  size\_t numNodes;  
  size\_t source;  
  size\_t sink;  
  size\_t iterator;  
  
  size\_t mFlow = 0;  
  
  bool stageIsChanged = 0;  
  bool debug = 0;  
  
  std::vector<Node> nodes;  
  std::vector<Edge> edges;  
  
  std::vector<std::vector<size\_t>> capacity;  
  std::vector<std::vector<size\_t>> residualCapacity;  
  std::vector<size\_t> deltaFlow = std::vector<size\_t>(1);  
  std::vector<size\_t> currentflows = std::vector<size\_t>(1);  
  
  std::vector<std::vector<std::vector<size\_t>>> flowEdge;  
  std::vector<std::vector<std::vector<size\_t>>> residualFlowEdge;  
  void readFile(const std::string &name) {  
    std::ifstream f(name);  
    if (!f.is\_open()) {  
      std::cout << "Error! File isn't opened\n";  
      exit(1);  
    }  
    for (std::string line; std::getline(f, line, '\n');) {  
      std::string buf;  
      std::stringstream line\_stream(line);  
      getline(line\_stream, buf, ' ');  
      if (buf == "node") {  
        std::getline(line\_stream, buf, ',');  
        size\_t id = std::stoi(buf);  
        std::getline(line\_stream, buf, ',');  
        size\_t x = std::stoi(buf);  
        std::getline(line\_stream, buf, ',');  
        size\_t y = std::stoi(buf);  
        nodes.push\_back({id, x, y});  
      } else if (buf == "edge") {  
        std::getline(line\_stream, buf, ',');  
        size\_t from = std::stoi(buf);  
        std::getline(line\_stream, buf, ',');  
        size\_t to = std::stoi(buf);  
        std::getline(line\_stream, buf, ',');  
        size\_t capacity = std::stoi(buf);  
        edges.push\_back({from, to, capacity});  
      } else if (buf == "source") {  
        std::getline(line\_stream, buf, ',');  
        source = std::stoi(buf);  
      } else if (buf == "sink") {  
        std::getline(line\_stream, buf, ',');  
        sink = std::stoi(buf);  
      }  
    }  
    f.close();  
  }  
  bool findAugmentingPath(size\_t source, size\_t sink,  
                          std::vector<size\_t> &parent) {  
    std::vector<bool> visited(numNodes, false);  
    std::queue<size\_t> q;  
    q.push(source);  
    visited[source] = true;  
    parent[source] = -1;  
    while (!q.empty()) {  
      auto u = q.front();  
      q.pop();  
      for (size\_t v = 0; v < numNodes; ++v) {  
        if (!visited[v] && residualCapacity[u][v] > 0) {  
          q.push(v);  
          parent[v] = u;  
          visited[v] = true;  
        }  
      }  
    }  
    auto table = std::vector<std::vector<size\_t>>(  
        nodes.size(), std::vector<size\_t>(nodes.size(), 0));  
  
    for (size\_t u = 0; u < numNodes; ++u) {  
      for (size\_t v = 0; v < numNodes; ++v) {  
        if (residualCapacity[u][v] < capacity[u][v]) {  
          table[u][v] = capacity[u][v] - residualCapacity[u][v];  
        }  
      }  
    }  
    flowEdge.push\_back(table);  
    residualFlowEdge.push\_back(residualCapacity);  
    return visited[sink];  
  }  
  
  void maxFlow() {  
    if (source == sink || source > numNodes - 1 || sink > numNodes - 1) {  
      return;  
    }  
    std::vector<size\_t> parent(numNodes, -1);  
  
    while (findAugmentingPath(source, sink, parent)) {  
      size\_t minCapacity = std::numeric\_limits<size\_t>::max();  
  
      for (size\_t v = sink; v != source; v = parent[v]) {  
        size\_t u = parent[v];  
        minCapacity = std::min(minCapacity, residualCapacity[u][v]);  
      }  
  
      deltaFlow.push\_back(minCapacity);  
  
      for (size\_t v = sink; v != source; v = parent[v]) {  
        size\_t u = parent[v];  
        residualCapacity[u][v] -= minCapacity;  
        residualCapacity[v][u] += minCapacity;  
      }  
  
      mFlow += minCapacity;  
    }  
    currentflows.resize(deltaFlow.size(), 0);  
    for (size\_t i = 1; i < currentflows.size(); i++) {  
      currentflows[i] = currentflows[i - 1] + deltaFlow[i];  
    }  
  }  
  void drawArrow(sf::RenderWindow &w, const sf::Vector2f &a,  
                 const sf::Vector2f &b, size\_t space, const sf::Color color) {  
    sf::Vector2f v = (b - a);  
    v /= hypot(v.x, v.y);  
    sf::Vector2f n = sf::Vector2f(v.y, -v.x);  
  
    auto q = sf::Vector2f(v.x \* (space - 2), v.y \* (space - 2));  
  
    auto t = b - q - sf::Vector2f(v.x \* 20, v.y \* 20);  
    auto t1 = t + sf::Vector2f(n.x \* 10, n.y \* 10);  
    auto t2 = t - sf::Vector2f(n.x \* 10, n.y \* 10);  
  
    sf::ConvexShape convex;  
  
    convex.setPointCount(4);  
    convex.setFillColor(color);  
  
    convex.setPoint(0, t1);  
    convex.setPoint(1, b - q);  
    convex.setPoint(2, t2);  
    convex.setPoint(3, b - sf::Vector2f(v.x \* 40, v.y \* 40));  
  
    w.draw(convex);  
  }  
  
public:  
  Graph(const std::string &file) {  
    readFile(file);  
    numNodes = nodes.size();  
    capacity.resize(numNodes, std::vector<size\_t>(numNodes, 0));  
    residualCapacity.resize(numNodes, std::vector<size\_t>(numNodes, 0));  
    for (const auto &edge : edges) {  
      this->capacity[edge.from][edge.to] = edge.capacity;  
      residualCapacity[edge.from][edge.to] = edge.capacity;  
    }  
  
    flowEdge = std::vector<std::vector<std::vector<size\_t>>>(  
        0, std::vector<std::vector<size\_t>>(  
               nodes.size(), std::vector<size\_t>(nodes.size(), 0)));  
    residualFlowEdge = std::vector<std::vector<std::vector<size\_t>>>(  
        0, std::vector<std::vector<size\_t>>(  
               nodes.size(), std::vector<size\_t>(nodes.size(), 0)));  
    maxFlow();  
  }  
  
  void drawGraph(sf::RenderWindow &window) {  
    size\_t radius = 30;  
    int textSpace = 15;  
  
    sf::Font font;  
    font.loadFromFile("tuffy.ttf");  
    for (auto &edge : edges) {  
      if (edge.from >= nodes.size() || edge.to >= nodes.size()) {  
        std::cerr << "Error: Edge index out of range\n";  
        continue;  
      }  
      sf::Color color = ColorEdgeDefault;  
  
      if (iterator) {  
        const auto &p = flowEdge[iterator];  
        for (int i = 0; i < static\_cast<int>(p.size()) - 1; ++i) {  
          auto flowBefore = flowEdge[(iterator - 1 + flowEdge.size()) %  
                                     (flowEdge.size())][edge.from][edge.to];  
          auto flowCurrent = flowEdge[iterator][edge.from][edge.to];  
          if (flowBefore < flowCurrent) {  
            color = ColorEdgeMore;  
  
            break;  
  
          } else if (flowBefore > flowCurrent) {  
            color = ColorEdgeLess;  
  
            break;  
          } else if (flowCurrent) {  
            color = ColorEdgeSame;  
          }  
        }  
      }  
  
      sf::Vector2f p1(static\_cast<float>(nodes[edge.from].x + radius),  
                      static\_cast<float>(nodes[edge.from].y + radius));  
      sf::Vector2f p2(static\_cast<float>(nodes[edge.to].x + radius),  
                      static\_cast<float>(nodes[edge.to].y + radius));  
  
      sf::Vector2f dir = p2 - p1;  
      sf::Vector2f perp(-dir.y, dir.x);  
      float length = sqrt(perp.x \* perp.x + perp.y \* perp.y);  
      perp /= length;  
  
      perp \*= 2.0f;  
  
      sf::ConvexShape line(4);  
      line.setPoint(0, p1 + perp);  
      line.setPoint(1, p2 + perp);  
      line.setPoint(2, p2 - perp);  
      line.setPoint(3, p1 - perp);  
      line.setFillColor(color);  
  
      window.draw(line);  
  
      drawArrow(window, p1, p2, radius, color);  
  
      sf::Text weightText;  
  
      auto cap = std::to\_string(edge.capacity);  
      if (iterator) {  
        auto flow = flowEdge[iterator][edge.from][edge.to];  
        cap = std::to\_string(flow) + "/" + cap;  
        if (debug) {  
          auto res1 = residualFlowEdge[iterator][edge.from][edge.to];  
          auto res2 = residualFlowEdge[iterator][edge.to][edge.from];  
          cap +=  
              "\nres: f" + std::to\_string(res1) + "/b" + std::to\_string(res2);  
        }  
      }  
      weightText.setString(cap);  
      weightText.setCharacterSize(15);  
      weightText.setFillColor(ColorTextFill);  
      weightText.setOutlineColor(ColorTextOutline);  
      weightText.setOutlineThickness(1.5);  
      weightText.setFont(font);  
      weightText.setStyle(sf::Text::Bold);  
  
      int k = 1;  
      if (p1.x > p2.x) {  
        k -= 2;  
      }  
      weightText.setPosition((p1.x + p2.x) / 2 - k \* textSpace,  
                             (p1.y + p2.y) / 2 - k \* textSpace);  
  
      window.draw(weightText);  
    }  
  
    for (const auto &node : nodes) {  
      auto color = ColorNodeDefault;  
      if (&nodes[source] == &node) {  
        color = ColorNodeSource;  
      }  
      if (&nodes[sink] == &node) {  
        color = ColorNodeSink;  
      }  
      sf::CircleShape circle(static\_cast<float>(radius));  
      circle.setFillColor(color);  
      circle.setOutlineColor(ColorTextOutline);  
      circle.setOutlineThickness(3);  
      circle.setPosition(static\_cast<float>(node.x),  
                         static\_cast<float>(node.y));  
      window.draw(circle);  
  
      sf::Text idText;  
      idText.setString(std::to\_string(node.id));  
      idText.setCharacterSize(30);  
      idText.setFillColor(ColorTextFill);  
      idText.setFont(font);  
      idText.setStyle(sf::Text::Bold);  
      idText.setOutlineColor(ColorTextOutline);  
      idText.setOutlineThickness(1.5);  
      idText.setPosition(static\_cast<float>(node.x + radius / 2 + 3),  
                         static\_cast<float>(node.y + radius / 2 + 3));  
      window.draw(idText);  
    }  
  
    {  
      sf::Text iterationText;  
      std::string flowText = "";  
  
      if (iterator) {  
        flowText = "iteration " + std::to\_string(iterator) +  
                   "\nflow = " + std::to\_string(currentflows[iterator]) +  
                   "\ndeltaFlow = " + std::to\_string(deltaFlow[iterator]);  
      }  
  
      flowText += "\nmax flow = " + std::to\_string(mFlow);  
      if (iterator == flowEdge.size() - 1) {  
        flowText = "Final " + flowText;  
      }  
      iterationText.setString(flowText);  
      iterationText.setCharacterSize(30);  
      iterationText.setFillColor(ColorTextOutline);  
      iterationText.setOutlineColor(ColorTextOutline);  
      iterationText.setOutlineThickness(0.5);  
      iterationText.setFont(font);  
      iterationText.setStyle(sf::Text::Italic);  
      iterationText.setPosition(580, 20);  
      window.draw(iterationText);  
    }  
  }  
  
  void changeStage(int step) {  
    stageIsChanged = true;  
    iterator = (iterator + step + flowEdge.size()) % (flowEdge.size());  
  }  
  void changeDebug() { debug = !debug; }  
  size\_t getMaxFlow() const { return mFlow; }  
  size\_t getSource() const { return source; }  
  size\_t getSink() const { return sink; }  
};  
  
int main() {  
  std::string name;  
  
  std::cout << "Graph to compute is ";  
  *//name = "graph1.txt";*  
  std::cin >> name;  
  sf::ContextSettings settings;  
  settings.antialiasingLevel = 8;  
  sf::RenderWindow window(sf::VideoMode(800, 600), name, sf::Style::Close,  
                          settings);  
  window.setVerticalSyncEnabled(true);  
  window.requestFocus();  
  
  Graph g(name);  
  
  std::cout << "Max flow from " << g.getSource() << " to " << g.getSink()  
            << " = " << g.getMaxFlow() << "\n";  
  
  while (window.isOpen()) {  
  
    sf::Event event;  
    while (window.pollEvent(event)) {  
      switch (event.type) {  
      case (sf::Event::Closed):  
        window.close();  
        break;  
      case (sf::Event::KeyReleased): {  
  
        switch (event.key.code) {  
        case (sf::Keyboard::Space):  
        case (sf::Keyboard::Enter):  
        case (sf::Keyboard::Right):  
          g.changeStage(1);  
          break;  
        case (sf::Keyboard::LControl):  
        case (sf::Keyboard::Left):  
          g.changeStage(-1);  
          break;  
        case (sf::Keyboard::LAlt):  
          g.changeDebug();  
          break;  
        default:  
          break;  
        }  
      }  
      case (sf::Event::MouseButtonReleased): {  
  
        switch (event.key.code) {  
        case (sf::Mouse::Left):  
          g.changeStage(1);  
          break;  
        case (sf::Mouse::Right):  
          g.changeStage(-1);  
          break;  
        default:  
          break;  
        }  
      }  
  
      default:  
        break;  
      }  
    }  
  
    window.clear(ColorBackground);  
  
    g.drawGraph(window);  
  
    window.display();  
  }  
  
  return 0;  
}

**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 | 0,3 | 0,8 | 22 неделя |
| Знакомство с литературой по теме курсовой работы | 2,7 | 7,5 | 24 неделя |
| Реализация отрисовки графа | 10 | 20 | 25 неделя |
| Реализация алгоритмов поиска пути | 5 | 20 | 27 неделя |
| Реализация алгоритма поиска максимального потока | 10 | 30 | 27 неделя |
| Исправление и улучшение кода программы | 5 | 13,4 | 30 неделя |
| Составление и оформление пояснительной записки и подготовка к защите | 2,7 | 7,5 | 40 неделя |
| Защита | 0,3 | 0,8 | 40 неделя |
| Итого | 36 | 100 |  |