**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ**

**УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»**

**(СПбГУТ)**

Факультет Инфокоммуникационных сетей и систем

Кафедра Защищенных систем связи

Дисциплина Разработка защищенных сетевых приложений

**Отчет по курсовой работе**

«Разработка программы по расчету минимального пути метрополитена СПб»

Студент:

Павлов К.К.

*(Ф.И.О., № группы) (подпись)*

Преподаватель:

ст. пр. ЗСС Цветков А.Ю.

*(уч. степень, уч. звание, Ф.И.О.) (подпись)*

I. Введение

2. Необходимость оптимизации путешествий в метро Санкт-Петербурга

B. Цель и задачи исследования

1. Разработка приложения для расчёта минимального пути в метро

2. Анализ и выбор оптимальных алгоритмов маршрутизации

C. Актуальность темы

1. Рост числа пассажиров и сложность навигации в городском транспорте

2. Потребность в эффективных решениях для повышения удобства перемещений

II. Обзор литературы

A. Анализ существующих методов расчёта минимального пути в графах

1. Обзор классических алгоритмов (Dijkstra, A\*, и т.д.)

2. Сравнительный анализ методов и их применимость к транспортным системам

B. Описание подходов к созданию приложений для навигации в транспортных системах

1. Изучение успешных примеров приложений для общественного транспорта

2. Выделение ключевых особенностей пользовательского опыта

III. Архитектура приложения

A. Выбор языка программирования (Java)

1. Обоснование выбора Java для разработки кроссплатформенного приложения

2. Преимущества использования Java в контексте данного проекта

B. Проектирование основных компонентов приложения

1. Выделение основных модулей (навигация, интерфейс, обработка данных)

2. Определение взаимодействия между компонентами

C. Используемые библиотеки и фреймворки

1. Выбор библиотек для упрощения разработки (например, JavaFX, Graphhopper)

2. Их роль в обеспечении функциональности приложения

IV. Моделирование графа метрополитена СПб

A. Определение узлов и связей

1. Географическое распределение станций метро

2. Учет особенностей пересадочных узлов

B. Взвешивание рёбер графа (время в пути, количество пересадок)

1. Анализ времени в пути между станциями

2. Определение весов для учета удобства перемещений

V. Алгоритмы расчёта минимального пути

A. Используемые алгоритмы

1. Разработка и обоснование выбора алгоритмов маршрутизации

2. Оптимизация алгоритмов для учета динамических изменений в системе

B. Реализация выбранных алгоритмов в контексте приложения

1. Кодирование алгоритмов для эффективного использования в мобильном приложении

2. Обработка исключительных ситуаций и обеспечение стабильной работы

VI. Разработка пользовательского интерфейса

A. Проектирование интерфейса приложения

1. Создание интуитивно понятного и удобного интерфейса пользователя

2. Учет требований к доступности и визуальному оформлению

B. Реализация взаимодействия с пользователем

1. Обработка ввода пользователя и предоставление информации

2. Разработка функционала для сохранения предпочтений пользователей

VII. Тестирование приложения

A. Модульное тестирование компонентов

1. Проверка корректности работы каждого модуля

2. Выявление и исправление ошибок в изолированных частях приложения

B. Интеграционное тестирование

1. Проверка взаимодействия между компонентами

2. Обеспечение совместной работы всех частей приложения

C. Оценка производительности

1. Измерение времени работы алгоритмов в различных условиях

2. Оптимизация производительности для обеспечения плавного пользовательского опыта

VIII. Заключение

A. Сводка результатов

1. Достижение целей и задач исследования

2. Выводы по эффективности разработанного приложения

B. Перспективы развития и улучшения приложения

1. Планы по добавлению новых функциональностей

2. Возможные области для дальнейших исследований

IX. Список использованных источников

A. Научные статьи, книги, онлайн-ресурсы

**Введение**

**Важность оптимизации маршрута в метро Санкт-Петербурга**

Оптимизация маршрута в метро Санкт-Петербурга является необходимой из-за ряда факторов, влияющих на эффективность и комфортность передвижения пассажиров. Вот несколько основных причин:

1. Экономия времени:

- Оптимизированные маршруты позволяют пассажирам добираться до места назначения быстрее, что особенно важно в условиях городского трафика и повышенной нагрузки на транспортную систему.

2. Снижение нагрузки на систему:

- Эффективное распределение пассажиропотока позволяет снизить перегрузку на перегонных участках и узких участках линий метро, что способствует улучшению общей производительности системы.

3. Экономия энергии:

- Оптимизированные маршруты позволяют снизить количество транспортных средств и, следовательно, энергопотребление, что может быть важным аспектом в условиях стремления к устойчивому развитию и снижению воздействия на окружающую среду.

4. Улучшение комфорта пассажиров:

- Оптимизация маршрутов может включать в себя установку дополнительных станций или пересадочных узлов в стратегических местах, что сделает поездки более удобными и сократит расстояние, которое пассажиры должны пройти для достижения своего пункта назначения.

5. Уменьшение заторов и задержек:

- Хорошо продуманные маршруты могут сократить вероятность возникновения заторов и задержек, обеспечивая более плавное движение по линиям метро и снижая риск чрезмерной нагрузки на станции.

6. Повышение безопасности:

- Оптимизированные маршруты также могут способствовать повышению безопасности, уменьшая количество случаев тесных пересадок или перегруженных поездов, что может привести к сокращению возможности инцидентов.

С учетом этих факторов оптимизация маршрутов в метро Санкт-Петербурга становится важным элементом улучшения общественного транспорта и обеспечения более эффективного и удобного передвижения горожан.

**Цели и задачи**

Цели проекта:

1. Анализ и выбор оптимальных алгоритмов маршрутизации.
2. Разработать функциональное приложение с целью облегчения навигации в метрополитене Санкт-Петербурга.

Задачи проекта:

1. Определить оптимальный алгоритм для подсчёта минимального пути и времени между вершинами графа.
2. Проанализировать особенности структуры и графа метрополитена Санкт-Петербурга
3. Определить узлы и связи графа, учитывая станции метро и пересадочные узлы.
4. Взвесить рёбра графа, учитывая время в пути и количество пересадок между станциями.
5. Выбрать язык программирования
6. Выбрать и реализовать алгоритмы маршрутизации, адаптированные для конкретных особенностей метрополитена.
7. Разработать архитектуру приложения, выделить основные модули и определить принципы их взаимодействия.
8. Обеспечить корректную работу алгоритмов в различных сценариях использования.
9. Реализовать функционал взаимодействия пользователя с приложением, включая ввод параметров маршрута и отображение результата.
10. Сформулировать выводы на основе результатов разработки и тестирования приложения.
11. Предложить возможные направления для дальнейшего совершенствования и расширения функционала.

**Задание курсовой работы**

Разработать программу по расчету минимального пути графа метрополитена СПб, используя матрицу смежности, характеризующую граф. Весом ребра графа является время, затрачиваемое для перемещения от одной станции к другой.

Программа должна получать на вход название станции отправления и название станции прибытия. По этим данным она рассчитывает минимальное время в пути и кратчайший маршрут между станциями.

**Анализ существующих методов расчёта минимального пути в графах**

**Алгоритм Дейкстры**

Алгоритм предназначен для нахождения кратчайших путей от одной из вершин графа до всех остальных вершин в графе, при условии, что веса рёбер неотрицательны. Этот алгоритм используется в случаях, когда необходимо определить оптимальный путь в сети с неотрицательными стоимостями перемещения между узлами.

Шаги работы алгоритма Дейкстры подробно:

1. Инициализация:

- Устанавливаем начальную вершину и присваиваем ей расстояние 0 (расстояние от начальной вершины до самой себя).

2. Присваивание начальных расстояний:

- Для всех остальных вершин графа устанавливаем начальные расстояния от начальной вершины до них как бесконечность. Временно создаем структуру данных (часто используется куча или очередь с приоритетом) для отслеживания текущих расстояний.

3. Основной цикл:

- На каждом шаге в основном цикле алгоритма выбирается вершина с наименьшим текущим расстоянием из созданной структуры данных. Начиная с этой вершины, рассматриваются все её соседи.

4. Релаксация рёбер:

- Для каждого соседа текущей вершины проверяем, существует ли путь через текущую вершину с меньшей стоимостью, чем текущее известное расстояние до этого соседа. Если существует, обновляем расстояние до соседа.

5. Пометка вершины:

- После рассмотрения всех соседей текущей вершины, помечаем её как посещённую.

6. Повторение:

- Повторяем шаги 3-5, пока не посетим все вершины графа.

По завершении алгоритма, для каждой вершины будет известно кратчайшее расстояние от начальной вершины. Если вам также нужны сами пути, можно сохранить информацию о предыдущих вершинах на пути к каждой вершине.

Преимущества алгоритма Дейкстры включают его простоту и эффективность для графов с неотрицательными весами рёбер. Однако он не подходит для графов с отрицательными весами и не обрабатывает циклы в графе.

**Алгоритм Беллмана-Форда**

Алгоритм используется для нахождения кратчайших путей от одной из вершин графа до всех остальных вершин, в том числе и в графах с рёбрами произвольного веса, включая те, которые могут иметь отрицательные стоимости. Однако алгоритм требует отсутствия циклов отрицательной стоимости, иначе он может зациклиться.

Шаги работы алгоритма Беллмана-Форда подробно:

1. Инициализация:

- Устанавливаем начальную вершину и присваиваем ей расстояние 0 (расстояние от начальной вершины до самой себя).

2. Присваивание начальных расстояний:

- Для всех остальных вершин графа устанавливаем начальные расстояния от начальной вершины до них как бесконечность.

3. Основной цикл:

- Повторяем V-1 раз, где V - количество вершин в графе.

- На каждой итерации алгоритма рассматриваем все рёбра графа и пытаемся улучшить оценки расстояний до вершин. Итерации нужны для того, чтобы учитывать все возможные пути к вершинам.

4. Релаксация рёбер:

- Для каждого ребра графа проверяем, существует ли путь через текущее ребро с меньшей стоимостью, чем текущее известное расстояние до вершины, находящейся на другом конце ребра. Если существует, обновляем расстояние до этой вершины.

5. Проверка наличия циклов отрицательной стоимости:

- После выполнения всех итераций проверяем граф на наличие циклов отрицательной стоимости. Если расстояния до вершин продолжают уменьшаться после выполнения V-1 итераций, то граф содержит цикл отрицательной стоимости.

6. Восстановление кратчайших путей (при необходимости):

- Если циклов отрицательной стоимости нет, можно восстановить кратчайшие пути, сохраняя информацию о предыдущих вершинах на каждой итерации алгоритма.

Преимущества алгоритма Беллмана-Форда включают его способность работать с графами с рёбрами произвольного веса, включая отрицательные веса, при условии отсутствия циклов отрицательной стоимости. Недостатком является его более высокая вычислительная сложность по сравнению с алгоритмом Дейкстры.

**Алгоритм Флойда-Уоршелла**

Алгоритм предназначен для нахождения кратчайших путей между всеми парами вершин в графе, даже если веса рёбер могут быть отрицательными. Этот алгоритм основан на динамическом программировании и выполняет итеративное обновление матрицы расстояний между парами вершин.

Описание работы алгоритма Флойда-Уоршелла:

1. Инициализация:

- Создаём матрицу расстояний D размерности VxV, где V - количество вершин в графе.

- Заполняем матрицу так, чтобы D[i][j] содержала вес ребра между вершинами i и j, если ребро существует, иначе D[i][j] устанавливается в бесконечность.

2. Основной цикл:

- Повторяем V раз (где V - количество вершин в графе). Каждая итерация обозначает добавление следующей вершины в множество промежуточных вершин.

3. Обновление матрицы расстояний:

- На каждой итерации рассматриваем все пары вершин i и j и проверяем, уменьшится ли расстояние от i до j, если мы используем промежуточную вершину k.

- Обновляем значение D[i][j] как минимум из текущего значения D[i][j] и суммы D[i][k] + D[k][j].

4. Восстановление кратчайших путей (при необходимости):

- Если нужно восстановить сами кратчайшие пути, сохраняем информацию о промежуточных вершинах при обновлении матрицы расстояний.

5. Проверка наличия отрицательных циклов:

- После завершения всех итераций проверяем матрицу расстояний на наличие отрицательных значений на её главной диагонали. Если такие значения есть, граф содержит отрицательный цикл.

Алгоритм Флойда-Уоршелла обладает преимуществом в том, что он находит кратчайшие пути между всеми парами вершин, что полезно в случае, когда нужно заранее знать все возможные кратчайшие пути. Однако его вычислительная сложность составляет O(V^3), что может быть неэффективным для больших графов.

**Алгоритм A\* (A-star)**

Алгоритм является эвристическим алгоритмом поиска кратчайшего пути в графе с весами на рёбрах. Этот алгоритм эффективен для поиска пути от начальной вершины к целевой вершине в графе. Алгоритм использует эвристическую функцию для приоритизации путей и улучшает производительность по сравнению с обычными алгоритмами поиска пути.

Вот подробное описание работы алгоритма A\*:

1. Инициализация:

- Устанавливаем начальную вершину и целевую вершину.

- Создаём два списков: открытый список (Open List) и закрытый список (Closed List).

- Помещаем начальную вершину в открытый список. Устанавливаем её стоимость g(x) равной 0 и вычисляем эвристическую функцию h(x) от начальной вершины до целевой вершины.

2. Основной цикл:

- Пока открытый список не пуст:

- Выбираем вершину с наименьшей суммой f(x) = g(x) + h(x) из открытого списка. Это вершина с наименьшей ожидаемой общей стоимостью.

- Перемещаем выбранную вершину из открытого списка в закрытый список.

- Если выбранная вершина - целевая вершина, то путь найден. Восстанавливаем путь, просматривая предыдущие вершины от целевой к начальной.

- Иначе рассматриваем всех соседей текущей вершины:

- Для каждого соседа:

- Если сосед уже в закрытом списке, игнорируем его.

- Если сосед не в открытом списке, добавляем его туда и вычисляем g(x), h(x) и f(x).

- Если сосед уже в открытом списке и новый путь короче, обновляем g(x) и f(x).

3. Завершение:

- Если открытый список опустел и целевая вершина не достигнута, то путь не существует.

Алгоритм A\* сочетает в себе эффективность алгоритма Дейкстры с использованием эвристической функции, что делает его особенно подходящим для поиска пути в больших графах. Однако важно правильно выбрать эвристическую функцию (h(x)), чтобы она была информативной и не приводила к некорректным результатам.

**Построение графа метро Санкт-Петербурга**

Граф — это математическая абстракция, представляющая собой набор вершин (узлов) и рёбер (связей) между этими вершинами. Графы используются для моделирования различных отношений и структур в различных областях, включая информатику, транспорт, социальные науки и другие.

Они бывают ориентированными, где рёбра имеют направление, и неориентированными, где связи двунаправленные. Графы применяются для анализа сетей, поиска кратчайших путей, оптимизации транспортных систем, моделирования социальных взаимодействий и многих других задач.

Использование графов позволяет компактно представлять сложные структуры и эффективно решать разнообразные задачи, связанные с взаимосвязями между элементами системы.

Матрица смежности графа — это квадратная матрица, в которой строки и столбцы соответствуют вершинам графа, а элементы матрицы указывают на наличие (или отсутствие) рёбер между соответствующими вершинами. Для неориентированного графа матрица смежности симметрична относительно своей главной диагонали.

Пример: если у нас есть граф с вершинами {A, B, C}, и рёбрами (A, B) и (B, C), то матрица смежности может выглядеть так:

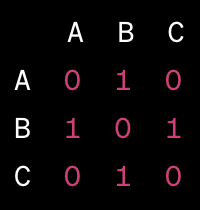


Рис. 1 Матрица смежности

В данном примере "1" в позиции (A, B) и (B, C) указывает на наличие рёбер между соответствующими вершинами.

Методы работы с матрицей смежности включают:

1. Поиск соседей вершины: Просмотр строки (или столбца) матрицы смежности позволяет определить соседние вершины для конкретной вершины.

2. Добавление и удаление рёбер: Манипуляции с элементами матрицы позволяют добавлять новые рёбра или удалять существующие.

3. Проверка существования ребра: Определение наличия ребра между двумя вершинами осуществляется проверкой соответствующего элемента матрицы.

4. Алгоритмы обхода графа: Матрица смежности часто используется для реализации алгоритмов обхода графа, таких как поиск в глубину (DFS) или поиск в ширину (BFS).

5. Представление графа: Матрица смежности предоставляет удобное представление графа, упрощая выполнение различных операций и алгоритмов.

Обходы матрицы смежности применяются для обхода графа и анализа его структуры. Два основных метода обхода графа в матрице смежности — это поиск в глубину (DFS) и поиск в ширину (BFS):

1. Поиск в глубину (DFS):

- Принцип работы: начинается с выбора начальной вершины и последующего поиска вглубь каждой ветви, пока не будет достигнут конец пути, затем возвращение и продолжение с другой ветви.

- Рекурсивная реализация: может быть реализован с использованием рекурсии, где для каждой вершины вызывается функция DFS для её соседей.

- Стек: может быть также реализован с использованием стека для отслеживания вершин и порядка обхода.

2. Поиск в ширину (BFS):

- Принцип работы: начинается с выбора начальной вершины, после чего посещаются все её соседи. Затем переходим к соседям по одному уровню от начальной вершины, затем к следующему уровню и так далее.

- Очередь: реализуется с использованием очереди для отслеживания порядка посещения вершин.

- Кратчайший путь: Поиск в ширину также часто используется для определения кратчайшего пути между двумя вершинами в невзвешенных графах.

Оба метода являются важными инструментами при работе с графами, и выбор между ними зависит от конкретных требований задачи. Поиск в глубину обычно применяется, если важно исследовать какую-то ветвь графа, в то время как поиск в ширину хорошо подходит для определения свойств графа на разных уровнях от начальной вершины.

Для построения графа был использован сайт для построения графов онлайн. Схема метро была перенесена в построитель. Станции являлись вершинами, а переезды и переходы между станциями – взвешенными ненаправленными рёбрами графа.

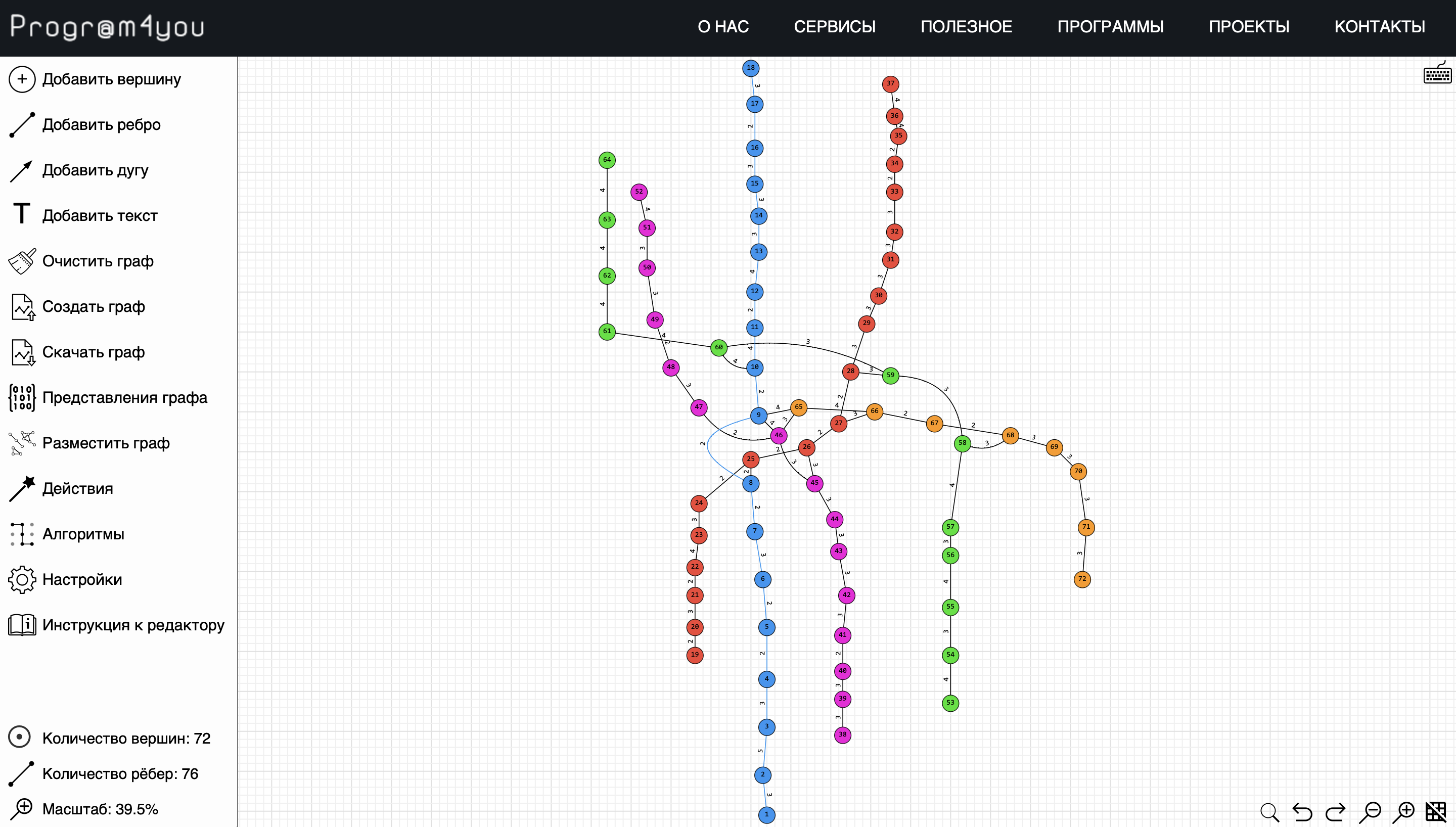


Рис. 2 - Граф метро Санкт-Петербурга

По данному графу была построена матрица смежности.

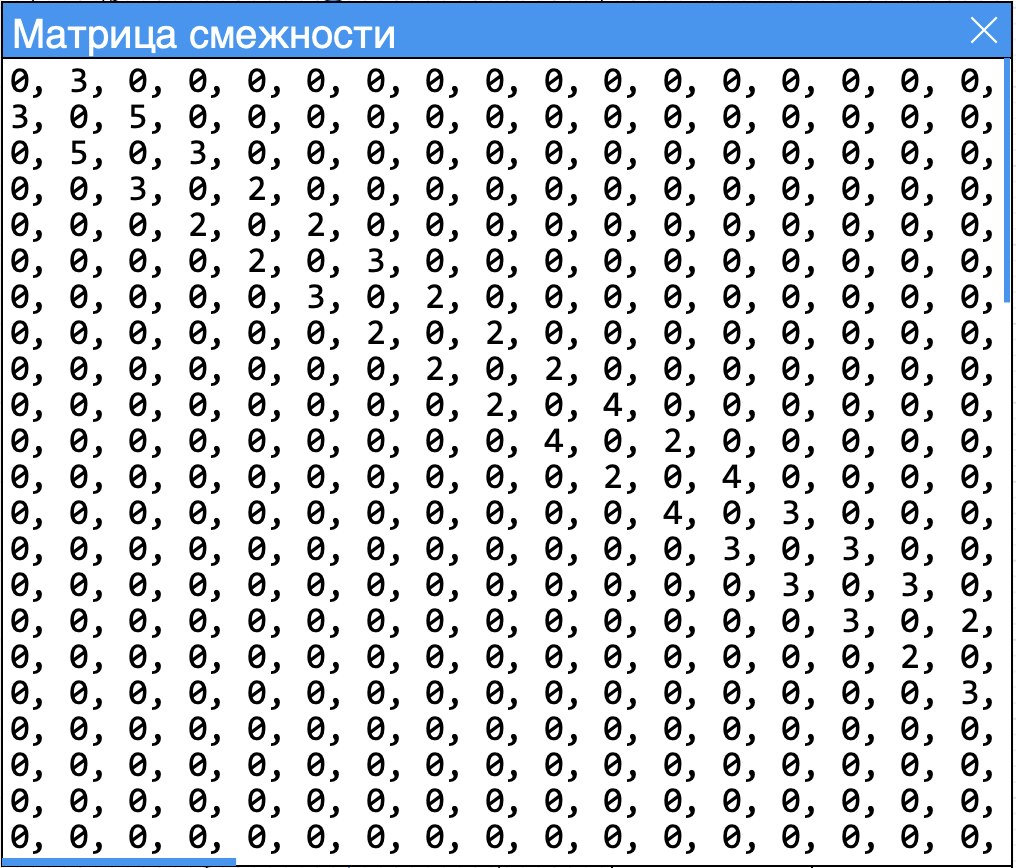


Рис. 3 - Матрица смежности метро СПб

**Язык программирования**

Java, созданная в 1996 году Джеймсом Гослингом и его командой в компании Sun Microsystems, представляет собой язык программирования, ориентированный на обеспечение переносимости кода между различными платформами. Основной идеей Java является принцип "Write Once, Run Anywhere" (WORA), что обеспечивает возможность написания кода единожды и его запуска на любой платформе, поддерживающей виртуальную машину Java (JVM).

Благодаря своей переносимости и простоте использования, Java стала популярной в различных областях, включая веб-разработку, мобильные приложения и серверные приложения. Одним из главных преимуществ этого языка является его платформонезависимость, обеспечивающая разработчикам удобство написания кода, который может выполняться на различных платформах без необходимости его адаптации.

Java также отличается простым и понятным синтаксисом, что делает ее доступной для изучения даже для новичков в программировании. Богатая библиотека классов предоставляет готовые решения для многих задач, что ускоряет процесс разработки. Высокая производительность и встроенные механизмы безопасности делают Java предпочтительным выбором для создания быстрых и эффективных приложений, особенно в крупных и сложных системах.

С релизом Java 20 язык продолжает свое развитие, предлагая новые функции и улучшения, что подчеркивает его актуальность и важность в современном программировании. Несмотря на появление новых технологий, Java остается одним из самых популярных языков программирования, широко используемым в различных сферах.

**Обоснование выбора Java для разработки кроссплатформенного приложения**

Выбор языка программирования для разработки кроссплатформенного приложения для расчета кратчайшего пути в метро Санкт-Петербурга зависит от различных факторов, таких как требования к проекту, опыт разработчиков, поддержка инструментов и технологий. Вот несколько аргументов в пользу выбора Java:

1. Кроссплатформенность:

- Java является кроссплатформенным языком программирования, что означает, что приложение, написанное на Java, может выполняться на различных операционных системах без изменений в исходном коде. Это особенно полезно для разработки приложений, которые должны работать на разных устройствах и платформах.

2. Большое сообщество и экосистема:

- Java имеет обширное сообщество разработчиков и обширную экосистему библиотек и фреймворков. Для разработки приложения для расчета кратчайшего пути в метро могут быть полезны различные библиотеки для работы с графами и алгоритмами поиска пути.

3. Надежность и производительность:

- Java известна своей стабильностью и надежностью. Виртуальная машина Java (JVM) обеспечивает управление памятью и выполнение кода, что способствует созданию стабильных и производительных приложений.

4. Инструменты для разработки:

- Java обладает множеством инструментов для разработки, от IDE (Integrated Development Environment) до средств тестирования и профилирования кода. Это может значительно упростить процесс разработки и обеспечить высокое качество кода.

5. Поддержка многопоточности:

- Работа с параллельными вычислениями и многопоточностью может быть важной частью при разработке приложения для расчета кратчайших путей в графе метро. Java имеет встроенную поддержку многопоточности, что может облегчить реализацию эффективных алгоритмов.

6. JavaFX для пользовательского интерфейса:

- Если планируется разработка графического интерфейса для взаимодействия с пользователем, JavaFX предоставляет удобные инструменты для создания кроссплатформенных GUI.

7. Большой опыт разработки приложений:

- Java широко используется в индустрии разработки программного обеспечения и имеет большой опыт в создании разнообразных приложений, включая крупные корпоративные системы.

**Сборка jar архива**

JAR (Java Archive) — это формат архива, используемый в среде разработки Java для упаковки и распространения Java классов и связанных ресурсов в один файл. Этот формат архива был введен с целью упрощения управления и развертывания Java приложений. Вот основные особенности и характеристики JAR архивов:

1. Упаковка классов и ресурсов:

- JAR архив позволяет объединить несколько файлов, включая скомпилированные классы, ресурсы, файлы конфигурации и библиотеки, в один компактный файл.

2. Формат ZIP:

- JAR архивы являются подмножеством формата ZIP. Это означает, что они используют структуру ZIP-архива для хранения файлов и поддерживают сжатие данных. Это сокращает размер JAR файла и улучшает эффективность его передачи по сети.

3. Манифест:

- Каждый JAR архив содержит файл манифеста (`META-INF/MANIFEST.MF`), который содержит метаинформацию об архиве. В манифесте могут быть определены различные атрибуты, такие как версия архива, главный класс для запуска приложения и другие настройки.

4. Запуск из командной строки:

- JAR архивы могут быть запущены из командной строки с использованием команды `java -jar YourJarFile.jar`. Это упрощает запуск Java приложений, так как JVM (виртуальная машина Java) автоматически обрабатывает все необходимые детали.

5. Зависимости и библиотеки:

- JAR архивы также могут включать в себя библиотеки и зависимости, что делает их удобными для распространения и использования в других проектах.

6. Поддержка многомодульных приложений:

- В JAR архивах можно организовать приложения с несколькими модулями, что обеспечивает логическую структуру проекта и упрощает его развертывание.

7. Применение в веб-разработке:

- JAR архивы также используются в веб-разработке, особенно при создании веб-приложений на платформе Java, таких как Java EE (Enterprise Edition).

8. Инструменты для создания и управления:

- Различные инструменты, такие как `jar` утилита, входящая в состав JDK (Java Development Kit), а также среды разработки, предоставляют средства для создания и управления JAR архивами.

Использование JAR архивов облегчает процесс разработки, тестирования и распространения Java приложений, делая их более переносимыми и удобными в использовании.

Основные шаги процесса сборки JAR архивов:

1. Написание исходного кода:

- Разработайте Java приложение и напишите необходимый исходный код.

2. Компиляция кода:

- Используйте Java компилятор (например, `javac`) для преобразования исходного кода в байт-код, который может быть выполнен виртуальной машиной Java (JVM).

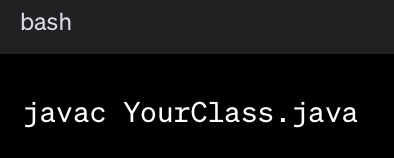


Рис. 4 - Компиляция кода

3. Создание файлов манифеста:

- Создайте файл манифеста, например, `Manifest.txt`, и определите в нем основные атрибуты.

Пример `Manifest.txt`:

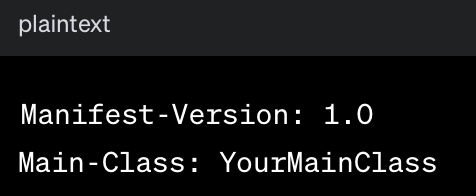


Рис. 5 - Создание манифеста

4. Упаковка в JAR:

- Используйте инструмент `jar`, поставляемый с JDK, для создания JAR архива. Укажите опции, включая манифест и имена файлов, которые вы хотите включить в JAR.

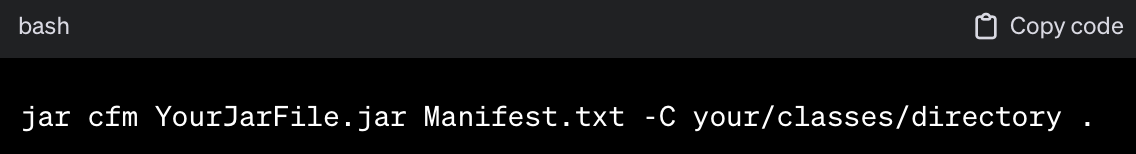


Рис. 6 - Упаковка в JAR

Здесь:

- `c` - создание нового архива.

- `m` - добавление манифеста.

- `f` - указание имени архива.

- `-C` - изменение каталога перед упаковкой файлов.

5. Проверка JAR:

- Проверьте созданный JAR архив с помощью команды `jar tf`.



Рис. 7 - Проверка JAR

Эта команда отобразит содержимое JAR архива.

6. Запуск JAR:

- Запустите JAR архив с помощью команды `java -jar`.

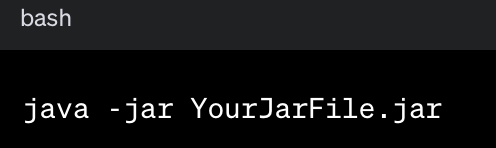


Рис. 8 - Запуск JAR

Эти шаги позволяют создавать, упаковывать и запускать Java приложения в виде JAR архивов. Однако, в зависимости от вашего проекта и среды разработки, процесс может немного отличаться. Например, многие современные среды разработки, такие как IntelliJ IDEA или Eclipse, предоставляют удобные средства для автоматизации сборки JAR архивов.

**Написание кода**

Процесс написания версии кода:

1. Определение структуры графа:

В данном случае, матрица смежности — это двумерный массив, где graph[i][j] содержит вес ребра между станциями i и j. Значение INF (бесконечность) используется для отсутствия прямой связи.

2. Определение методов для расчета минимального пути:

Создаётся метод `dijkstra`, который будет реализовывать алгоритм Дейкстры. Этот метод должен принимать матрицу смежности, начальную станцию и рассчитывать кратчайший путь до всех остальных станций, а затем рассчитывать минимальное время и оптимальный путь.

3. Добавление метода для вывода результата:

Создаётся метод `printSolution`, который будет выводить минимальное время и оптимальный путь в консоль. Для вывода кратчайшего пути используется стек, который заполняется пройденными станциями и выводится в обратном порядке. Этот метод может использоваться для отладки и тестирования.

4. Реализация метода поиска индекса станции с минимальным временем из не посещённых станций:

Создаётся метод `minDistance`, в котором возвращаемое значение:

Индекс станции с минимальным временем из непосещенных. Если все станции посещены, возвращается -1.

Логика метода:

Инциализируются переменные min и minIndex значением Integer.MAX\_VALUE и -1 соответственно.

Проход по всем станциям в массиве distance.

Если станция еще не посещена (!visited[v]) и время до нее меньше текущего минимального времени (distance[v] < min), обновляем min и minIndex.

Возвращаем индекс станции с минимальным временем из не посещённых.

Этот метод используется внутри алгоритма Дейкстры для выбора станции с наименьшим временем в каждом шаге обновления расстояний.

5.Хранение справочных данных:

Для удобства работы все справочные данные были помещены в класс под названием `GraphMatrix`. В нём будет храниться матрица смежности графа, словари с названиями станций для обработки введённых названий.

6. Написание основного класса `Main`:

Основной класс `Main` содержит обработку введённых пользователем названий станций и достаёт из словаря индекс станции в матрице смежности. Для обработки ситуаций если пользователь неправильно ввёл название станции прописано условие сравнения названия станции со словарём, и программа выведет пользователю сообщение о том, что название введено неверно. Так же класс содержит вызов алгоритма Дейкстры и последующий вывод результата работы программы в консоль.

7. Тестирование:

Запустить программу и убедитесь, что она правильно рассчитывает минимальное время и выводит оптимальный путь в консоль.

Приведенный выше процесс ориентирован на консольное приложение без использования интерфейса.

**Примеры работы приложения**

При корректном вводе станции отправления и прибытия программа выведет минимальное время в пути и кратчайший маршрут:

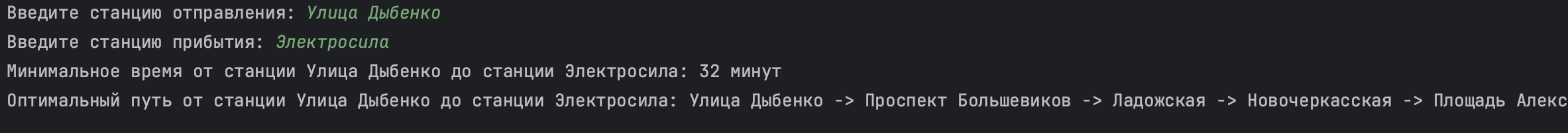


Рис. 9 - Результат успешной работы программы

При вводе одной и той же станции в графу отправления и прибытия программа так же выведет минимальное время, которое будет равняться 0:

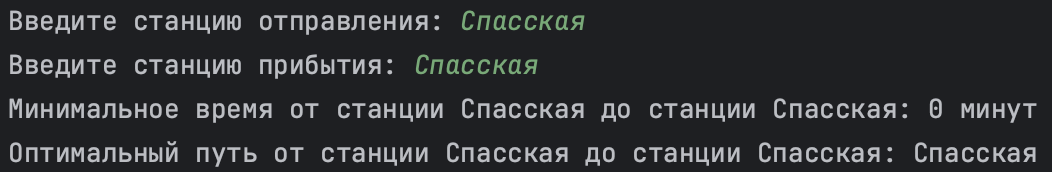


Рис. 10 - Результат успешной работы программы при одинаковых станциях

Если пользователь ошибся при вводе названия станции отправления, то программа сообщит о том, что название введено неверно:

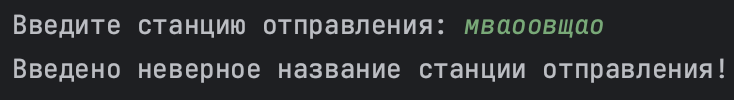


Рис. 11 - Результат работы программы при вводе некорректных данных в графу названия станции отправления

Если пользователь ошибся при вводе названия станции прибытия, то программа сообщит о том, что название введено неверно:

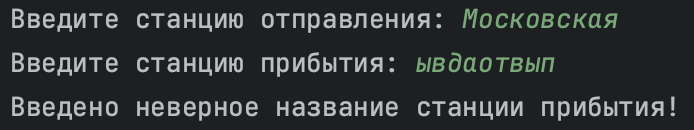


Рис. 12 - Результат работы программы при вводе некорректных данных в графу названия станции прибытия

**Написание JavaDoc**

JavaDoc - это инструмент для автоматической генерации документации из комментариев в исходном коде на языке программирования Java. Он позволяет программистам создавать структурированные и понятные описания классов, методов, полей и других элементов кода, что упрощает процесс документирования программного обеспечения.

Структура комментариев:

Комментарии для JavaDoc должны начинаться с /\*\* и завершаться \*/. В начале каждого файла и каждого отдельного класса следует поместить комментарий с общим описанием, используя теги @author и @version. Для каждого метода следует добавить комментарий с описанием функциональности, используя тег @param для параметров и @return для возвращаемого значения.

Описание классов и методов:

Используйте тег @param для описания параметров метода. Используйте тег @return для описания возвращаемого значения. Добавляйте теги, такие как @throws для описания исключений, которые метод может выбрасывать.

Использование HTML-тегов:

Можно использовать HTML-теги в комментариях для форматирования текста в документации. Например, для создания списка используйте <ul> и <li>.

Генерация в HTML формате производится через фунцию генерация JavaDoc в IntelliJ IDEA.

**Блок схема алгоритма**

Блок-схема — это графическое представление алгоритма или программы с использованием блоков, представляющих различные этапы выполнения, и стрелок, обозначающих поток управления между этими блоками. Блок-схемы используются для визуализации логики выполнения программы и облегчения понимания ее структуры.

Основные элементы, которые часто включаются в блок-схемы программ:

Прямоугольники (блоки): представляют действия или операции, выполняемые программой. Например, это может быть блок для ввода данных, блок для выполнения определенного вычисления или блок для вывода результата.

Ромбы (условия): показывают ветвления в программе в зависимости от определенных условий. Например, это может быть проверка значения переменной и ветвление на основе результата этой проверки.

Овалы (начало/конец): обозначают начало и конец программы. Ввод данных обычно начинается с овала, а вывод данных или завершение программы - также с овала.

Стрелки: показывают направление потока управления между блоками. Обычно стрелки указывают на направление выполнения программы.

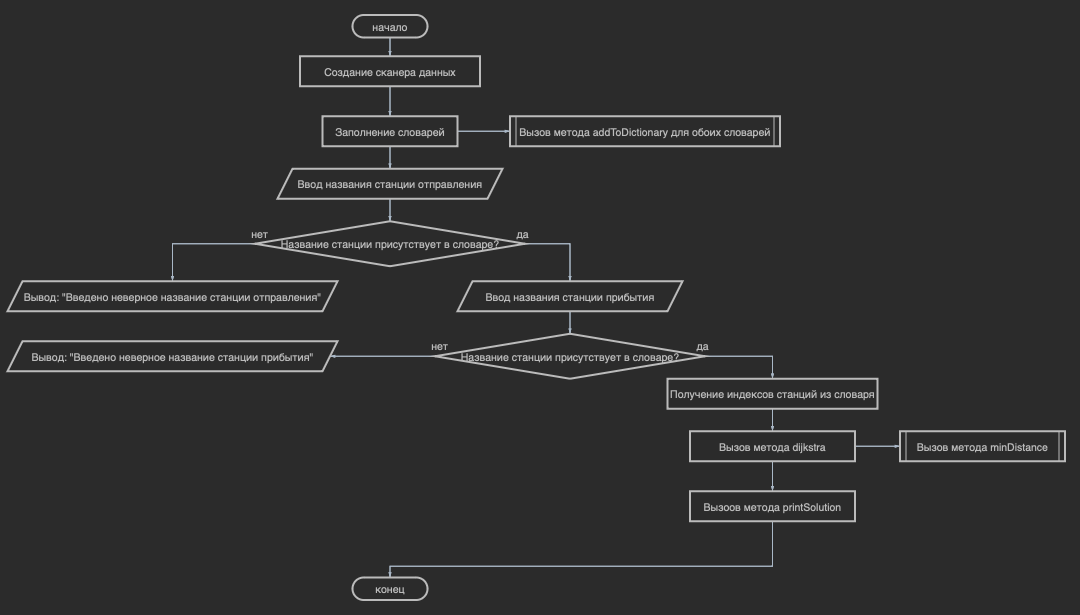


Рис. 13 - Блок схема программы