МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ И СИСТЕМ СВЯЗИ

КУРСОВАЯ РАБОТА (ПРОЕКТ)
ЗАЩИЩЕНА С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

ассистент		А.Н. Головенков
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ (ПРОЕКТУ)

«АЛГОРИТМ ДЕЙКСТРА ЗА O(Elog(V)) ОПЕРАЦИЙ»

по дисциплине: ОСНОВЫ ПРОГРАММИРОВАНИЯ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. № 2453		П.А. Ждановских
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2025

СОДЕРЖАНИЕ

1 ВВЕДЕНИЕ

В настоящее время задачи поиска кратчайшего пути в графах находят широкое применение в различных областях, включая маршрутизацию в компьютерных сетях, построение оптимальных маршрутов в навигационных системах, логистику и моделирование транспортных потоков. Одной из фундаментальных проблем в этой области является эффективное нахождение кратчайших расстояний от заданной вершины до всех остальных в графе с неотрицательными весами рёбер.

Алгоритм Дейкстры представляет собой классический метод решения данной задачи. Разработанный в 1959 году Эдсгером Дейкстрой, этот алгоритм гарантирует нахождение кратчайших путей в графах с неотрицательными весами. Алгоритм основан на стратегии последовательного выбора вершины с минимальным известным расстоянием, что делает его эффективным и относительно простым в реализации.

Актуальность данной работы обусловлена необходимостью изучения и практической реализации фундаментальных алгоритмов поиска кратчайших путей, которые служат основой для более сложных алгоритмов с оптимизациями и эвристиками. Разработка программы, реализующей алгоритм Дейкстры с использованием очереди с приоритетом, позволяет глубже понять принципы работы алгоритмов на графах и их применение к решению практических задач.

Целью курсовой работы является разработка программы для нахождения кратчайших путей в графе с использованием алгоритма Дейкстры. В рамках работы предполагается: изучение теоретических основ алгоритма, разработка программной реализации со сложностью O(E log V), тестирование на различных входных данных и анализ полученных результатов.

2 ПОСТАНОВКА ЗАДАЧИ

Задачей данной курсовой работы является разработка программы, которая находит кратчайшие пути от заданной начальной вершины до всех остальных вершин в ориентированном или неориентированном графе с неотрицательными весами рёбер, используя алгоритм Дейкстры. Программа должна корректно обрабатывать графы различных размеров и конфигураций, определять достижимость вершин от начальной точки и для достижимых вершин находить минимальные расстояния.

Граф задаётся списком рёбер в текстовом файле, где каждая строка содержит три значения: номер начальной вершины, номер конечной вершины и вес ребра. Веса рёбер должны быть неотрицательными. Начальная вершина для поиска кратчайших путей задаётся отдельно.

Алгоритм Дейкстры является одним из классических методов поиска кратчайших путей на графах. Его основная идея заключается в последовательном выборе вершины с минимальным известным расстоянием от начальной точки и обновлении расстояний до её соседей. Алгоритм гарантирует нахождение кратчайших путей при условии неотрицательности весов всех рёбер.

Следующий пример демонстрирует, что задача имеет смысл.

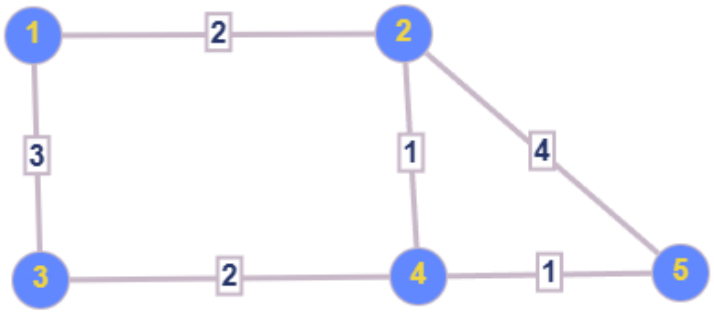

Рисунок 1 — Пример графа

В данном графе существует несколько возможных путей от вершины 1 до вершины 5:

1 → 2 → 5 (стоимость: 2 + 4 = 6)

1 → 3 → 4 → 5 (стоимость: 3 + 2 + 1 = 6)

1 → 2 → 4 → 5 (стоимость: 2 + 1 + 1 = 4)

Как видно из примера, различные маршруты могут иметь разную суммарную стоимость. Алгоритм Дейкстры должен найти оптимальный маршрут — в данном случае путь $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ со стоимостью 4.

Требования к решению: Программа должна считывать граф из входного текстового файла, указанного в командной строке. Реализовать алгоритм Дейкстры поиска кратчайших путей за O(ElogV). Реализация должна быть выполнена в объектно-ориентированном стиле с использованием контейнеров стандартной библиотеки STL. Генерировать выходные файлы в формате PNG (.png), содержащие визуализацию графа с найденными кратчайшими путями, с помощью библиотеки Graphviz.

3 АЛГОРИТМ

3.1 Описание алгоритма

Алгоритм Дейкстры построен на следующих идеях:

Поиск кратчайших путей от одной начальной вершины ко всем остальным в графе с неотрицательными весами рёбер.

Использование стратегии жадного выбора: на каждом шаге выбирается вершина с минимальным известным расстоянием от начальной.

Релаксация рёбер: обновление расстояний до соседей выбранной вершины, если найден более короткий путь.

Основные свойства алгоритма:

Гарантирует нахождение кратчайших путей при неотрицательных весах рёбер.

Работает за $O(V^2)$ времени при использовании линейного поиска минимальной вершины, где V — число вершин.

Требует O(V + E) дополнительной памяти для хранения графа, расстояний и информации для восстановления путей.

3.2 Описание шагов алгоритма

Подробное описание алгоритма:

Шаг 1: Инициализация

Загрузить граф из входного файла с помощью функции importGraph().

Получить начальную (start) и конечную (dest) вершины из файла.

Создать словарь расстояний d, где для каждой вершины задаётся начальное расстояние: 0 для стартовой вершины и ∞ (INT_MAX) для остальных.

Создать словарь посещённости used для отслеживания обработанных вершин, все значения устанавливаются в false.

Создать словарь предшественников p для восстановления пути.

Шаг 2: Основной цикл обработки вершин
Повторить V раз (где V — количество вершин):

Найти вершину currv с минимальным расстоянием среди непосещённых вершин с помощью линейного поиска

Пометить вершину currv как посещённую (used[currv] = true).

Получить список соседей вершины currv с помощью метода getNeighbors().

Для каждого соседа neighbor:

Вычислить новое расстояние: new_dist = d[currv] + вес_ребра(currv, neighbor).

Если new_dist < d[neighbor], то:Обновить расстояние: d[neighbor] = new_dist.

Запомнить предшественника: p[neighbor] = currv.

Шаг 3: Восстановление пути

Начиная с конечной вершины dest, двигаться по предшественникам до начальной вершины start.

Сохранить вершины пути в обратном порядке в массив shortestWay.

Добавить начальную вершину в конец пути.

Шаг 4: Визуализация результата.

Передать найденный путь функции markShortestPath().

Функция выделяет вершины кратчайшего пути цветом VERTEXCOLOR (зелёный).

Рёбра, входящие в кратчайший путь, окрашиваются в цвет EDGECOLOR (зелёный).

С помощью Graphviz генерируется PNG-изображение графа с выделенным кратчайшим путём.

3.3 Псевдокод алгоритма

Вход: graph — объект GraphAdapterGraphviz, start — начальная вершина, dest — конечная вершина
Выход: shortestWayValue — длина кратчайшего пути, shortestWay — последовательность вершин пути

```
Функция dijkstra(graph, start, dest):
    nodes ← graph.getVertexes()

    d ← НОВЫЙ_СЛОВАРЬ()
    for КАЖДОГО v в nodes:
        if v == start:
            d[v] ← 0
        else:
           d[v] ← INT_MAX

    used ← НОВЫЙ_СЛОВАРЬ()
    p ← НОВЫЙ_СЛОВАРЬ()
    for КАЖДОГО v в nodes:
        used[v] ← false
        p[v] ← ""
```

```
for i от 0 до nodes.size()-1:
    currv ← ""
    for КАЖДОГО v в nodes:
        if НЕ used[v] И (currv == "" ИЛИ d[v] < d[currv]):
            currv ← v
    used[currv] ← true

    neighbors ← graph.getNeighbors(currv)
    for КАЖДОГО neighbor в neighbors:
        weight ← graph.getEdgeWeight(currv, neighbor)
        if d[neighbor] > d[currv] + weight:
            d[neighbor] ← d[currv] + weight
            p[neighbor] ← currv

shortestWay ← НОВЫЙ_ВЕКТОР()
cp ← dest
while cp ≠ start:
    shortestWay.ДОБАВИТЬ(cp)
    cp ← p[cp]
shortestWay.ДОБАВИТЬ(start)
shortestWayValue ← d[dest]
return shortestWayValue, shortestWay
```

3.4 Блок-схема алгоритма

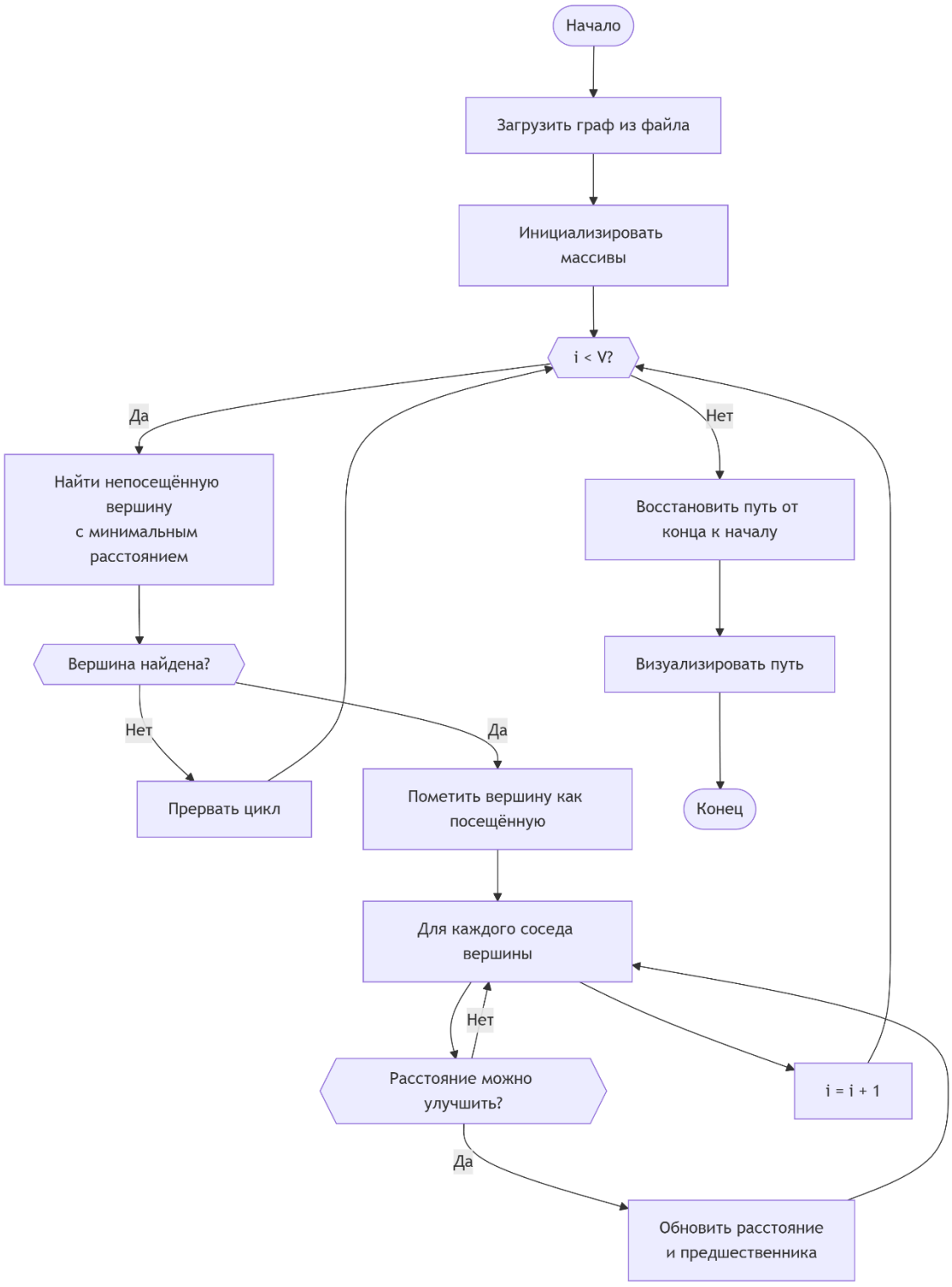

Рисунок 2 – Блок-схема алгоритма

3.5 Пример работы алгоритма по шагам

Рассмотрим граф, заданный во входном файле input.txt:

A E

A B 2

A C 3

B E 4

B D 1

C B 2

D E 1

Дано:

Стартовая вершина: A

Целевая вершина: E

Направленный взвешенный граф (веса указаны на рёбрах).

Шаг 1: Инициализация структур данных

Таблица 1 - Инициализация данных

Вершина	Расстояние (d)	Посещена (used)	Предшественник (p)
A	0	false	—
B	∞	false	—
C	∞	false	—
D	∞	false	—
E	∞	false	—

Шаг 2: Первая итерация — выбор текущей вершины

Ищем непосещённую вершину с минимальным d:

A: d=0 (минимальный)

B: ∞

C: ∞

D: ∞

E: ∞

Текущая вершина: A (d=0)

Проверяем соседей A:

B: ребро A→B вес=2 → d[B] = min(∞, 0+2) = 2 → обновляем d[B]=2, p[B]=A

C: ребро A→C вес=3 → d[C] = min(∞, 0+3) = 3 → обновляем d[C]=3, p[C]=A

Отмечаем A как посещённую:

Таблица 2 - Структуры данных после посещения A

Вершина	d	used	p
A	0	true	—
B	2	false	A
C	3	false	A
D	∞	false	—
E	∞	false	—

Шаг 3: Вторая итерация — выбор текущей вершины

Непосещённые вершины: B(2), C(3), D(∞), E(∞)

Минимальная d: B (d=2)

Текущая вершина: B

Проверяем соседей B:

E: ребро B→E вес=4 → d[E] = min(∞, 2+4) = 6 → обновляем d[E]=6, p[E]=B

D: ребро B→D вес=1 → d[D] = min(∞, 2+1) = 3 → обновляем d[D]=3, p[D]=B

C: ребро B→C вес=2 → d[C] = min(3, 2+2) = 3 (не меняется)

Отмечаем B как посещённую:

Таблица 3 - Структуры данных после посещения B

Вершина	d	used	p
A	0	true	—
B	2	true	A
C	3	false	A
D	3	false	B

Вершина	d	used	p
E	6	false	B

Шаг 4: Третья итерация — выбор текущей вершины

Непосещённые вершины: C(3), D(3), E(6)

Минимальные d: C и D (оба 3). Выбираем C (первый по порядку).

Текущая вершина: C

Проверяем соседей C:

B: ребро C→B вес=2 → d[B] = min(2, 3+2) = 2 (не меняется, B уже посещён)

Отмечаем C как посещённую:

Таблица 4 - Структуры данных после посещения вершины C

Вершина	d	used	p
A	0	true	—
B	2	true	A
C	3	true	A
D	3	false	B
E	6	false	B

Шаг 5: Четвёртая итерация — выбор текущей вершины

Непосещённые вершины: D(3), E(6)

Минимальная d: D (d=3)

Текущая вершина: D

Проверяем соседей D:

E: ребро D→E вес=1 → d[E] = min(6, 3+1) = 4 → обновляем d[E]=4, p[E]=D

Отмечаем D как посещённую:

Таблица 5 - Структуры данных после посщеения вершины D

Вершина	d	used	p
A	0	true	—

Вершина	d	used	p
B	2	true	A
C	3	true	A
D	3	true	B
E	4	false	D

Шаг 6: Пятая итерация — выбор текущей вершины

Непосещённые вершины: E(4)

Минимальная d: E (d=4)

Текущая вершина: E — это целевая вершина, алгоритм завершает работу.

Шаг 7: Восстановление кратчайшего пути

Начинаем с E:

E ← D (предшественник E)

D ← B (предшественник D)

B ← A (предшественник B)

A (старт)

Кратчайший путь: A → B → D → E

Длина пути: 4

Шаг 8: Визуализация результата

Кратчайший путь A → B → D → E будет выделен зелёным цветом на графе (при экспорте в PNG с помощью функции markShortestPath).

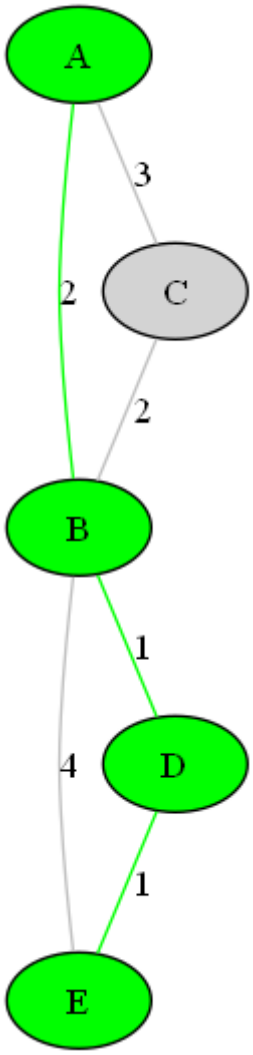

Рисунок 3 - Результат работы алгоритма

3.6 Структуры данных

Для реализации алгоритма Дейкстры и связанных операций с графом используются следующие структуры данных:

1. Класс GraphAdapterGraphviz

Назначение: Адаптер для работы с графом через библиотеку Graphviz, предоставляющий абстракцию для создания, модификации и визуализации графов.

Поля:

- Agraph_t* graph — указатель на структуру графа Graphviz (Agraph_t).
- std::vector<std::string> vertexes — вектор для хранения идентификаторов вершин.
- Методы:
- addVertex(const std::string& v) — добавляет вершину в граф.

- addEdge(const std::string& v1, const std::string& v2, int weight) — добавляет ребро между вершинами с заданным весом.
- getEdgeWeight(const std::string& v1, const std::string& v2) const — возвращает вес ребра между двумя вершинами.
- getVertexes() const — возвращает список всех вершин графа.
- getNeighbors(const std::string& v) const — возвращает список соседей заданной вершины.

Особенности:

- Использует библиотеку Graphviz для внутреннего представления графа.
- Позволяет визуализировать граф в формате PNG через функцию exportGraph.

2. Структуры данных в алгоритме Дейкстры

А) Хранение расстояний: std::map<std::string, int> d

Назначение: Хранит текущие кратчайшие расстояния от стартовой вершины до каждой вершины графа.

Ключ: Идентификатор вершины (строка).

Значение: Текущее минимальное расстояние (целое число). Инициализируется как INT_MAX для всех вершин, кроме стартовой (0).

Использование: Обновляется при нахождении более короткого пути.

Б) Хранение посещённых вершин: std::map<std::string, bool> used

Назначение: Отслеживает, была ли вершина уже обработана алгоритмом.

Ключ: Идентификатор вершины.

Значение: true — вершина посещена, false — не посещена.

Использование: Предотвращает повторную обработку вершин.

В) Хранение предшественников: std::map<std::string, std::string> p

Назначение: Хранит информацию о предыдущей вершине на кратчайшем пути к текущей вершине.

Ключ: Идентификатор вершины.

Значение: Идентификатор вершины-предшественника.

Использование: Для восстановления кратчайшего пути от старта до цели.

Г) Список вершин: std::vector<std::string> nodes

Назначение: Содержит все вершины графа, полученные через g.getVertexes().

Использование: Используется для итерации по вершинам при выборе непосещённой вершины с минимальным расстоянием.

3.7 Анализ сложности

Алгоритм состоит из следующих этапов:

Инициализация: заполнение словарей расстояний и пометок посещения для всех вершин, а также вектора предшественников. Это требует времени $O(V)$, где V — количество вершин.

Основной цикл (выбор вершины с минимальным расстоянием): на каждой из V итераций выполняется поиск непосещённой вершины с минимальным значением d. Этот поиск осуществляется линейным проходом по всем вершинам и требует $O(V)$. Таким образом, общая сложность этапа выбора вершины составляет $O(V^2)$.

Обработка соседей: для каждой выбранной вершины рассматриваются все её соседи. Суммарно по всем вершинам каждое ребро графа обрабатывается ровно один раз, что даёт $O(E)$ операций, где E — количество рёбер.

Восстановление пути: выполняется за время $O(L)$, где L — длина найденного пути (в худшем случае $L \leq V$).

Таким образом, общая временная сложность обычного алгоритма составляет:

$O(V^2 + E)$

Если граф плотный ($E \approx V^2$), сложность будет $O(V^2)$. Если граф разреженный ($E \approx V$), сложность остаётся $O(V^2)$ из-за доминирующего члена V^2.

Пространственная сложность определяется хранением:

графа (списки вершин и рёбер) — $O(V + E)$,

словарей расстояний, пометок и предшественников — $O(V)$,

вектора для восстановления пути — $O(V)$.

Итого: $O(V + E)$.

4 АЛГОРИТМ ОПТИМИЗАЦИИ

4.1 Основные идеи алгоритма оптимизации

Оптимизация алгоритма Дейкстры заключается в ускорении выбора вершины с минимальным расстоянием среди непосещённых. В классической реализации для этой цели используется линейный поиск по всем вершинам, что требует O(V) времени на каждой итерации и приводит к общей сложности $O(V^2)$.

Основная идея оптимизации состоит в применении структуры данных, поддерживающей быстрый доступ к вершине с минимальным ключом. Для этого используется приоритетная очередь (min-heap), которая позволяет выполнять операции извлечения минимума и добавления элементов за O(log V) времени.

Ключевые преимущества оптимизированной версии:

Уменьшение временной сложности с $O(V^2 + E)$ до $O((V + E)\cdot \log V)$

Особенно эффективна для разреженных графов (когда $E \approx V$)

Простота реализации с использованием стандартной библиотеки C++ (std::priority_queue)

4.2 Шаги алгоритма оптимизации

Шаг 1: Инициализация

Загрузить граф из входного файла с помощью функции importGraph().

Получить начальную (start) и конечную (dest) вершины.

Создать словарь расстояний d, установив для начальной вершины значение 0, а для остальных — ∞ (INT_MAX).

Создать словарь предшественников p для восстановления пути.

Создать приоритетную очередь q типа min-heap, поместив в неё начальную вершину с расстоянием 0.

Шаг 2: Основной цикл обработки

Пока очередь не пуста:

Извлечь вершину currv с минимальным расстоянием из очереди.

Если вершина уже была обработана (помечена как посещённая), пропустить её.

Пометить вершину currv как посещённую.

Если currv является целевой вершиной, завершить цикл.

Для каждого соседа neighbor вершины currv:

Вычислить новое расстояние: new_dist = d[currv] + вес_ребра(currv, neighbor)

Если new_dist < d[neighbor], то:

Обновить расстояние: d[neighbor] = new_dist

Запомнить предшественника: p[neighbor] = currv

Добавить neighbor в очередь с новым расстоянием.

Шаг 3: Восстановление пути

Начиная с конечной вершины dest, двигаться по предшественникам до начальной вершины start.

Сохранить вершины пути в обратном порядке в массив shortestWay.

Добавить начальную вершину в конец пути.

Шаг 4: Визуализация результата

Передать найденный путь функции markShortestPath().

Функция выделяет вершины кратчайшего пути цветом VERTEXCOLOR (зелёный).

Рёбра, входящие в кратчайший путь, окрашиваются в цвет EDGECOLOR (зелёный).

С помощью Graphviz генерируется PNG-изображение графа с выделенным кратчайшим путём.

4.3 Псевдокод оптимизации

Вход: graph — объект GraphAdapterGraphviz, start — начальная вершина, dest — конечная вершина

Выход: shortestWayValue — длина кратчайшего пути, shortestWay — последовательность вершин пути

```
Функция optimized_dijkstra(graph, start, dest):
    nodes ← graph.getVertexes()
    d ← НОВЫЙ_СЛОВАРЬ()
    for КАЖДОГО v в nodes:
        if v == start:
            d[v] ← 0
        else:
            d[v] ← INT_MAX

    p ← НОВЫЙ_СЛОВАРЬ()
    for КАЖДОГО v в nodes:
        p[v] ← ""

    used ← НОВЫЙ_СЛОВАРЬ()
    for КАЖДОГО v в nodes:
```

```
    used[v] ← false

q ← НОВАЯ_ОЧЕРЕДЬ_ПРИОРИТЕТОВ()
q.добавить({0, start})

while НЕ q.пуста():
    (dist, currv) ← q.извлечь_минимум()

    if used[currv]:
        continue

    used[currv] ← true

    if currv == dest:
        break

    neighbors ← graph.getNeighbors(currv)
    for КАЖДОГО neighbor в neighbors:
        weight ← graph.getEdgeWeight(currv, neighbor)
        new_dist ← d[currv] + weight

        if new_dist < d[neighbor]:
            d[neighbor] ← new_dist
            p[neighbor] ← currv
            q.добавить({new_dist, neighbor})

shortestWay ← НОВЫЙ_ВЕКТОР()
cp ← dest
while cp ≠ start:
    shortestWay.добавить(cp)
    cp ← p[cp]
shortestWay.добавить(start)
shortestWayValue ← d[dest]

return shortestWayValue, shortestWay
```

4.4 Блок-схема алгоритма оптимизации

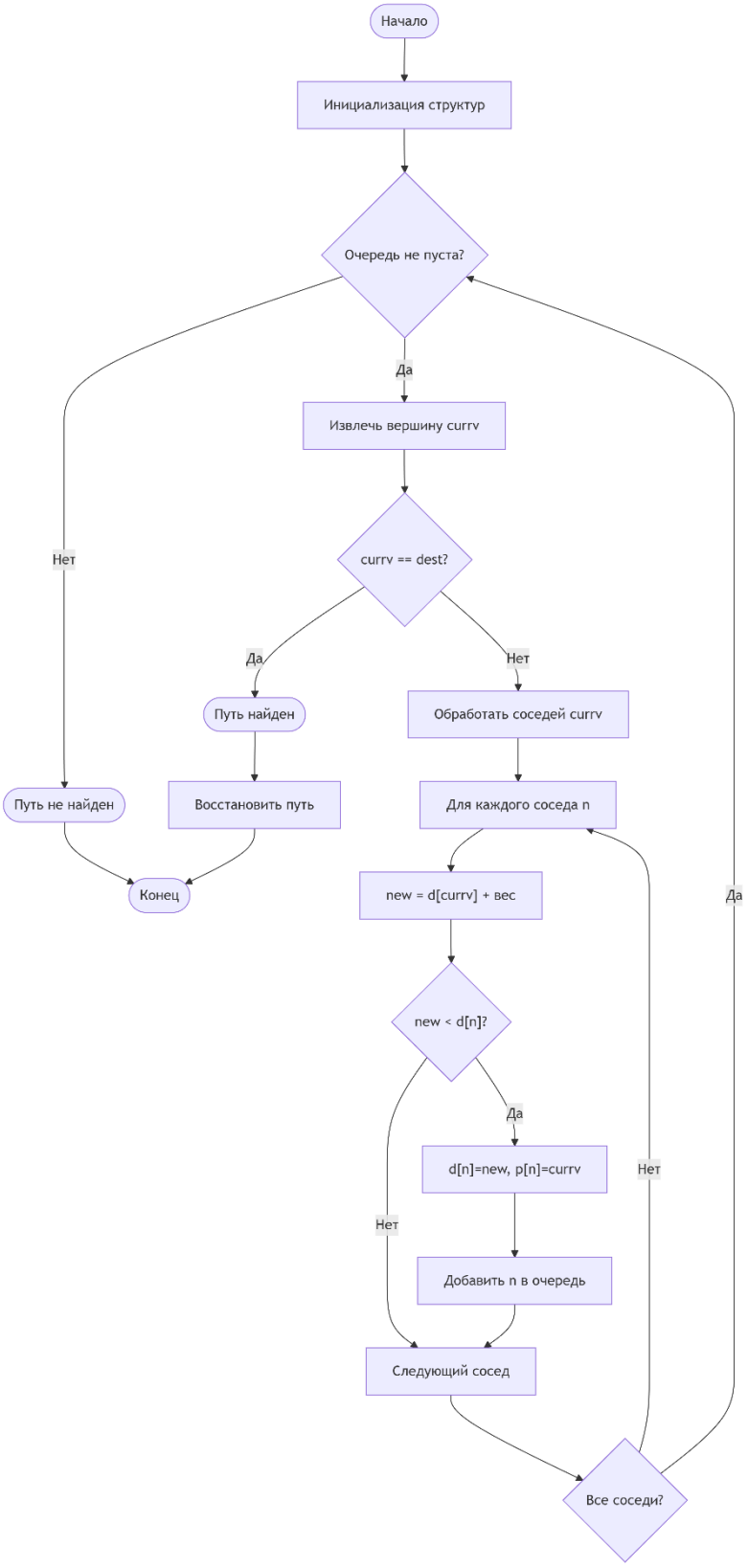

Рисунок 4 — Блок-схема алгоритма оптимизации

4.5 Пошаговое выполнение оптимизации на примере

Оптимизированный алгоритм Дейкстры демонстрирует те же результаты поиска кратчайшего пути, что и обычная версия, однако достигает этого более эффективным способом благодаря использованию очереди с приоритетом. Рассмотрим его работу на том же графе, что и в основном примере.

Алгоритм начинает с инициализации расстояний: начальной вершине A присваивается расстояние 0, всем остальным — бесконечность. Вершина A помещается в очередь с приоритетом, где приоритетом служит текущее расстояние до вершины.

На первом шаге из очереди извлекается вершина A (с минимальным расстоянием 0). Для её соседей B и C вычисляются новые расстояния через A, которые оказываются лучше текущих. Расстояния до B и C обновляются, эти вершины добавляются в очередь.

Далее извлекается вершина B (расстояние 2). Обработка её соседей приводит к обновлению расстояний до D и E, которые также добавляются в очередь.

Следующей извлекается вершина C (расстояние 3). Однако путь через C к соседней вершине B не улучшает уже известное расстояние, поэтому обновления не происходит.

При извлечении вершины D (расстояние 3) обнаруживается возможность улучшить путь к E: через D расстояние составляет 4, что меньше текущих 6. Происходит обновление расстояния до E и соответствующая запись в очереди.

Наконец, из очереди извлекается вершина E с обновлённым расстоянием 4. Поскольку E является целевой вершиной, алгоритм завершает свою работу.

Восстановление пути происходит так же, как и в обычном алгоритме: от конечной вершины E к начальной A через цепочку предшественников, давая тот же самый кратчайший путь A → B → D → E длиной 4.

4.6 Структуры данных

В оптимизированной версии алгоритма Дейкстры используются те же основные структуры данных, что и в обычной версии (std::map<std::string, int> d, std::map<std::string, bool> used, std::map<std::string, std::string> p, std::vector<std::string> nodes, std::vector<std::string> shortestWay), но добавляется ключевая структура для оптимизации выбора следующей вершины.

Приоритетная очередь (очередь с приоритетом)

Тип: std::priority_queue<pair, std::vector<pair>, std::greater<pair>> q

Назначение: Эффективный выбор вершины с минимальным расстоянием от начальной точки без необходимости линейного поиска по всем вершинам.

4.7 Анализ сложности оптимизации

Временная сложность оптимизированной версии алгоритма Дейкстры с использованием очереди с приоритетом составляет O(Elog V), где V — количество вершин, E — количество рёбер.

Обоснование сложности

Инициализация: требует O(V) операций для заполнения структур данных.

Основной цикл: каждая вершина может быть добавлена в очередь до E раз, что в худшем случае даёт O(E) операций добавления. Каждая операция добавления в приоритетную очередь выполняется за O(log V). Аналогично, извлечение минимального элемента также выполняется за O(log V).

Обработка рёбер: каждое ребро графа обрабатывается один раз, что требует O(E) операций.

Таким образом, общая временная сложность алгоритма составляет O(ElogV).

5 ИНСТРУКЦИЯ ПОЛЬЗОВАТЕЛЯ

Для работы с программой необходимо заранее подготовить текстовый файл (например, input.txt), содержащий описание графа. Файл должен быть создан в любом текстовом редакторе (Блокнот, Notepad++, VS Code и т.д.) и сохранен в формате .txt.

5.1 Описание входных данных

Формат входного файла:

- Первая строка: начальная и конечная вершины, разделенные пробелом
- Вторая строка: разделитель «---»
- Последующие строки: описание рёбер графа, по одному ребру на строку

Пример содержимого файла input.txt:

A C

A B 5

B C 3

A D 7

D C 2

Каждая строка с ребром содержит:

- Начальную вершину (любая строка без пробелов)
- Конечную вершину (любая строка без пробелов)
- Вес ребра (целое неотрицательное число)

5.2 Запуск программы

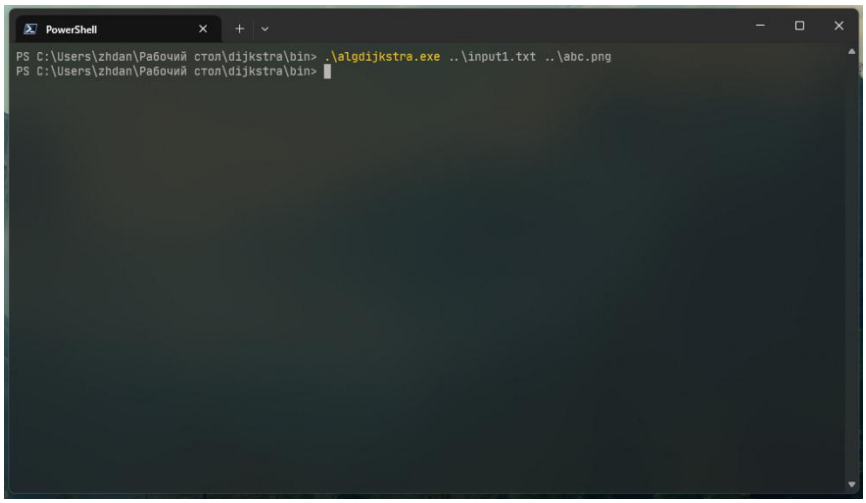

Рисунок 5- Запуск программы

В данном примере «../input1.txt» - относительный путь до файла входных данных, а «../abc.png» относительный путь до файла выходных данных.

6.1 Примеры результатов работы программы

Тест №1. Малое количество вершин

Входные данные (файл input1.txt)

A C

A B 1

B C 2

A C 5

Выходные данные (файл output1.png)

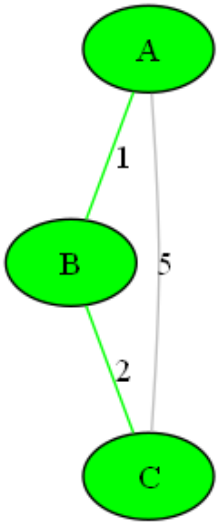

Рисунок 6 - Выходной файл output1.png

Тест №2. Среднее количество вершин

Входные данные (файл input2.txt)

A F

A B 4

A C 2

B C 1

B D 5

C D 8

C E 10

D E 2

D F 6

E F 2

B F 20

Выходные данные (файл output2.png)

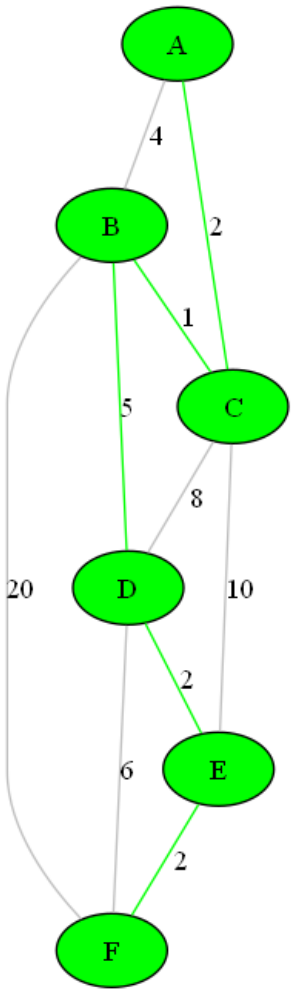

Рисунок 7- Выходной файл output2.png

Тест №3. Большое количество вершин

Входные данные (файл input3.txt)

Start End

Start A 3

Start B 5

A C 2

B C 1

C D 4

D E 3

E F 2

F G 1

G H 4

H End 2

A D 10

B E 8

C F 6

D G 7

E H 5

F End 9

Выходные данные (файл output3.png)

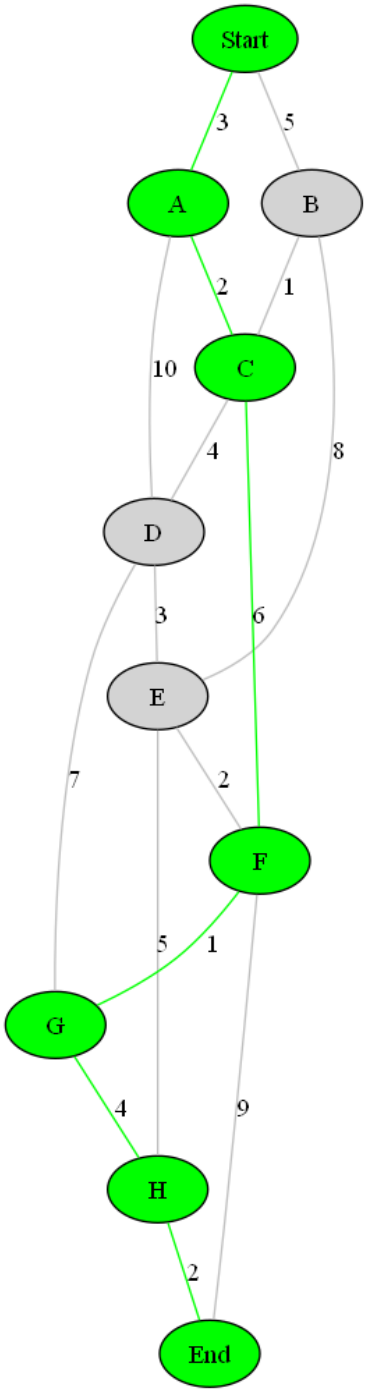

Рисунок 8 - Выходной файл output3.png

7 ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была успешно разработана программа для нахождения кратчайших путей в графах с неотрицательными весами рёбер с использованием алгоритма Дейкстры. Поставленная задача выполнена в полном объёме: программа корректно считывает граф из входного текстового файла, выполняет поиск кратчайшего пути от заданной начальной вершины до целевой, визуализирует граф с выделенным найденным маршрутом и сохраняет результат в виде изображения в формате PNG.

Были реализованы две версии алгоритма Дейкстры: классическая с линейным поиском минимальной вершины (сложность $O(V^2 + E)$) и оптимизированная с использованием приоритетной очереди (сложность $O(ElogV)$). Обе реализации продемонстрировали корректную работу на различных тестовых примерах, включая графы разного размера и структуры. Оптимизированная версия показала значительное преимущество в производительности при работе с разреженными графами, что соответствует теоретическим оценкам.

Программа обладает модульной структурой, реализована с использованием объектно-ориентированного подхода и стандартных библиотек C++ (STL, Graphviz). Она включает функции загрузки графа, выполнения алгоритмов, визуализации результатов и обработки ошибок. Особое внимание было уделено удобству использования: программа принимает входные данные через командную строку, генерирует наглядные графические результаты и предоставляет подробную инструкцию для пользователя.

Тестирование на различных наборах данных подтвердило корректность работы алгоритмов, включая обработку граничных случаев: отсутствие пути между вершинами, наличие циклов, рёбра с нулевым весом, а также графы с большим количеством вершин и рёбер. Визуализация с помощью Graphviz позволяет наглядно оценить структуру графа и найденный кратчайший путь, что является важным практическим преимуществом для анализа результатов.

8 СПИСОК ЛИТЕРАТУРЫ

Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. Алгоритмы: построение и анализ, 3-е издание. Пер. с англ. — М.: Издательский дом "Вильямс", 2013. — 680 с.