Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных технологий

Кафедра программного обеспечения вычислительной техники и

автоматизированных систем

**КУРСОВАЯ РАБОТА**

по дисциплине «Алгоритмы и структуры данных»

**Использование алгоритмов обработки данных при разработке приложений**

Пояснительная записка

ОГУ 09.03.04.3023.НОМЕР-ЗАЧЕТКИ ПЗ

|  |
| --- |
| Руководитель  старший преподаватель  С.А. Климачев  « » 2023 г.  Студент группы 22ПИнж(б)РПИС-1  К.Р. Ягудина  « » 2023 г. |

Оренбург 2023

Утверждаю

заведующий кафедрой программного

обеспечения вычислительной техники

и автоматизированных систем

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Д.В. Горбачев

подпись

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2023 г.

**ЗАДАНИЕ**

**на выполнение курсовой работы**

студенту Федорову Илье Васильевичу

(фамилия имя отчество)

по направлению подготовки 09.03.04 Программная инженерия

код, наименование

по дисциплине «Алгоритмы и структуры данных»

наименование дисциплины, модуля

1 Тема работы «Использование алгоритмов обработки данных при разработке приложений»

2 Срок сдачи студентом работы «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г.

3 Цель и задачи работы овладение навыками использования структур данных и алгоритмов обработки данных при разработке приложений; закрепление знаний и развитие практических навыков разработки программных средств.

4 Исходные данные к работе Вариант №5.

НАПИСАТЬ СВОЕ ЗАДАНИЕ

5 Перечень вопросов, подлежащих разработке 1) анализ структур данных и алгоритмов их обработки; 2) разработка алгоритмов программного средства; 3) программная реализация алгоритмов и структур данных; 4) анализ эффективности алгоритма решения задачи; 5) тестирование программного средства.

Дата выдачи и получения задания (30.10)

Руководитель «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_ г. \_\_\_\_\_\_\_\_\_\_\_\_С.А. Климачев

Студент «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_ г. \_\_\_\_\_\_\_\_\_\_\_\_\_\_И.В. Федоров

**Аннотация**

Данная курсовая работа посвящена разработке программы для построения веломаршрутов с использованием различных типов дорог и местности. В работе осуществлено исследование принципов построения веломаршрутов на основе карты местности и соединения точек маршрута различными типами полос движения.

Первая глава работы представляет основные теоретические аспекты построения веломаршрутов, включая цели, особенности и этапы выполнения программы, а также возможные варианты типов дорог.

Во второй главе проведена разработка программной реализации построения веломаршрутов, включая алгоритм работы программного средства и детали для работы с картами и различными типами дорог.

Третья глава представляет функции программы, ответственные за демонстрацию построенных веломаршрутов. Предоставлена таблица с перечнем этих функций и их описанием.

Четвертая глава посвящена анализу временной сложности программы построения веломаршрутов. Приводятся соответствующие формулы и выводы.

Пятая глава содержит результаты тестирования программного обеспечения, демонстрирующего построение веломаршрутов на различных типах местности и дорог.Общий объем пояснительной записки к курсовой работе составляет 29 листов текста, включая 9 рисунков, 6 таблиц, 2 формулы и 1 приложение

**Содержание**

[Введение 5](#_Toc154629045)

[1 Анализ структур данных и алгоритмов их обработки 6](#_Toc154629046)

[1.1 Граф 6](#_Toc154629047)

[2 Разработка алгоритмов программного средства 10](#_Toc154629048)

[3 Программная реализация структур данных 16](#_Toc154629049)

[4 Анализ эффективности алгоритма решения задачи 16](#_Toc154629050)

[5 Тестирование программного средства 19](#_Toc154629051)

[Заключение 21](#_Toc154629052)

[Список использованных источников 22](#_Toc154629053)

[Приложение А 23](#_Toc154629054)

**Введение**

В контексте современных информационных технологий особенно важна потребность в предоставлении студентам и специалистам возможности получать высококачественное образование и развивать свои навыки в области алгоритмов и графов. Один из фундаментальных элементов в этом обучении - освоение принципов работы алгоритмов на графах, таких как алгоритм Дейкстры.

Цель данной курсовой работы заключается в овладении навыками использования структур данных и алгоритмов обработки данных для разработки приложений, а именно создание инновационного обучающего программного средства. Это средство предназначено для демонстрации работы алгоритма Дейкстры с возможностью регулирования параметров, таких как ограничение на пропускную способность ребер. Такое программное средство позволит пользователям лучше понять основные принципы работы данного алгоритма и освоить его применение в практических задачах, связанных с графами.

Для достижения цели работы необходимо решить следующие инженерные задачи:

Описать алгоритм Дейкстры, используемый в программном средстве, его принципы функционирования, особенности и способы реализации. Разработать алгоритм работы программного средства, алгоритм решения задачи;

Разработать алгоритм работы программного средства, включая алгоритм решения задачи нахождения кратчайших путей в графе. Выполнить тестирование разработанного программного средства;

Разработать программное средство с удобным пользовательским интерфейсом, предусматривающее обработку исключений и некорректных действий пользователя.

Выполнить тестирование разработанного программного средства.

Актуальность данной работы обусловлена необходимостью обеспечения эффективного обучения алгоритмам на графах, которые являются важной составляющей информационных технологий. Развитие обучающих программ, визуализирующих алгоритмы на графах, способствует повышению качества образования в области программирования.

В рамках курсовой работы будут рассмотрены методологические основы разработки обучающих программ, проведен анализ теоретических исследований в области алгоритмов на графах, исследованы особенности алгоритма Дейкстры.

Практическая значимость данной работы заключается в возможности использования разработанной обучающей программы в учебном процессе, а также для самостоятельного изучения алгоритмов на графах студентами и специалистами в области информационных технологий.

# 1 Анализ структур данных и алгоритмов их обработки

## 1.1 Граф

Графы — это структуры данных, представляющие собой набор вершин, соединенных рёбрами. Для эффективного нахождения кратчайших путей в графе используется алгоритм Дейкстры, который может быть оптимизирован с использованием бинарной кучи. На Рисунке 1 представлен пример графа. На рисунке 1 представлен пример графа[1].

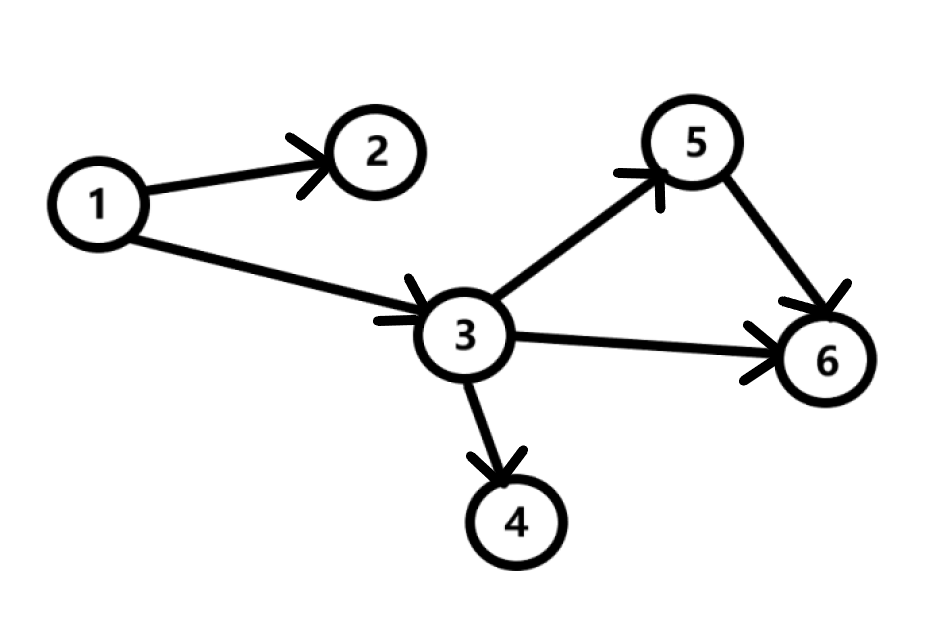


Рисунок 1 – граф

Граф может быть представлен в виде списка смежности или матрицы смежности. В контексте алгоритма Дейкстры, каждой вершине графа присваивается временная метка, представляющая текущую оценку расстояния от начальной вершины до данной вершины.

Основные понятия:

Вершины и Рёбра: Граф состоит из вершин и рёбер, где рёбра соединяют вершины.[2].

Веса ребер: Каждому ребру приписан вес, представляющий стоимость прохождения по этому ребру.[1-4].

Операции графа

1 Вставка вершины: Вставка вершины: Добавление новой вершины в граф. Если вершина еще не существует в графе, создается новая вершина с уникальным идентификатором, и ей присваивается новый список смежности. Новая вершина добавляется в открытое место на самом нижнем уровне списка.[5].

2 Вставка ребра: Добавление нового ребра между двумя вершинами. Ребро добавляется в список смежности соответствующих вершин, обеспечивая связь между ними. [5].

3 Удаление вершины: Удаление вершины из графа. Включает в себя удаление всех рёбер, связанных с данной вершиной, а также её удаление из списков смежности других вершин. После этого может потребоваться перестройка графа.

4 Удаление ребра: Удаление ребра между двумя вершинами. Ребро удаляется из соответствующего списка смежности, разрывая связь между вершинами.

Графы используются для реализации алгоритма Дейкстры. Алгоритм Дейкстры представляет собой эффективный метод поиска кратчайших путей от одной начальной вершины ко всем остальным вершинам во взешенном графе.[10]. Он использует подход, основанный на жадности, для выбора наименьшего пути на каждом этапе, обеспечивая минимизацию суммарных весов путей. [3].

Алгоритм Дейкстры представляет собой мощный инструмент для решения задач поиска кратчайших путей и находит применение в различных областях, где важна оптимизация маршрутов, например построения маршрута между двумя точками в городе.[7]. На рисунке 2 представлен пример работы алгоритма Дейкстры.

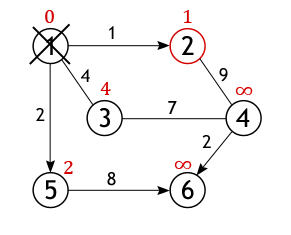


Рисунок 2 – Алгоритм Дейкстры для графа

Алгоритм Дейкстры можно разбить на 4 пункта:

1) Инициализация Пусть вершиной, из которой мы будем считать маршруты, будет 0. Расстояние до самой себя у этой вершины логично равно нулю. Остальные мы пока не знаем, поэтому отметим символом бесконечности. [6].

2) Первый шаг алгоритма. Мы выбираем еще не посещенную вершину с самой маленькой меткой относительно исходной — то есть такую, которая находится ближе всех. На первом шаге это одна из соседних вершин — та, которая соединена с исходной самым «маленьким» ребром. [6].

3) Дальнейшие шаги алгоритма. Для выбранной точки нужно осмотреть соседей и записать длину пути до них с учетом пройденного пути. А потом выбрать ближнюю точку. Но есть нюанс: нужно учитывать точки, которые мы уже использовали в прошлый раз. Если они дают более «выгодный» путь, лучше воспользоваться ими.

4) Конец алгоритма. Когда непосещённые вершины заканчиваются, алгоритм прекращает работу. Результат его действия — список кратчайших маршрутов до каждой точки из исходной. Для каждого маршрута указана его длина.

Псевдокод для поиска кратчайшего пути в графе:

FUNCTION Dijkstra(graph G, startVertex, endVertex, acceptRoads):

distances = {}

visited = {}

pq = PriorityQueue()

for vertex in G:

distances[vertex] = infinity

visited[vertex] = False

distances[startVertex] = 0

pq.insert(pair(0, startVertex))

while not pq.empty():

currentVertex = pq.extract\_min().second

if visited[currentVertex]:

continue

visited[currentVertex] = True

for edge in G.outgoing\_edges(currentVertex):

neighbor = edge.get\_destination\_vertex\_id()

newDistance = distances[currentVertex] + edge.get\_weight()

if edge.getRoad() != accept\_roads:

continue

if newDistance < distances[neighbor]:

distances[neighbor] = newDistance

pq.insert(pair(distances[neighbor], neighbor))

Инициализация контейнеров для посещенных вершин и меток расстояния от стартовой вершины представляет собой начальный этап. Затем все вершины перебираются, и для каждой из них смотрятся соседи, добавляя их в приоритетную очередь pq. Это играет ключевую роль в выборе следующей вершины для рассмотрения. Пары (расстояние, вершина) в очереди упорядочиваются так, чтобы вершины с наименьшими текущими расстояниями стояли впереди. Затем алгоритм переходит к основному циклу. На каждой итерации извлекается вершина с минимальным расстоянием currentVertex из приоритетной очереди. Если данная вершина уже посещена, она пропускается, иначе помечается как посещенная. Далее алгоритм рассматривает все исходящие ребра из текущей вершины и обновляет расстояния до соседей, если найден более короткий путь. Такой подход позволяет эффективно итерироваться по вершинам в порядке возрастания расстояний, выбирая каждый раз ближайшую вершину для дальнейшего рассмотрения. При использовании приоритетной очереди алгоритм Дейкстры оптимизирован для нахождения кратчайших путей в графе.

# 2 Разработка алгоритмов программного средства

Это описание структуры программы говорит о том, что она использует консольный интерфейс для взаимодействия с пользователем. Каждое действие пользователя инициирует определенную последовательность операций, включая инициализацию данных, выполнение соответствующих действий и отображение результатов.

Например, если пользователь выбирает опцию "1" для добавления вершины, программа запускает соответствующий код, который, вероятно, предлагает пользователю ввести данные для новой вершины и затем обновляет граф соответствующим образом.

Также, если пользователь выбирает опцию "d" для работы с алгоритмом Дейкстры и графами, программа переключается на соответствующую логику обработки данных графа, вероятно, предоставляя возможность выбора начальной и конечной вершин для выполнения алгоритма Дейкстры.

Этот подход делает программу более гибкой и удобной в использовании, поскольку предоставляет пользователю понятные сценарии взаимодействия и позволяет ему легко выполнять нужные операции с графом.

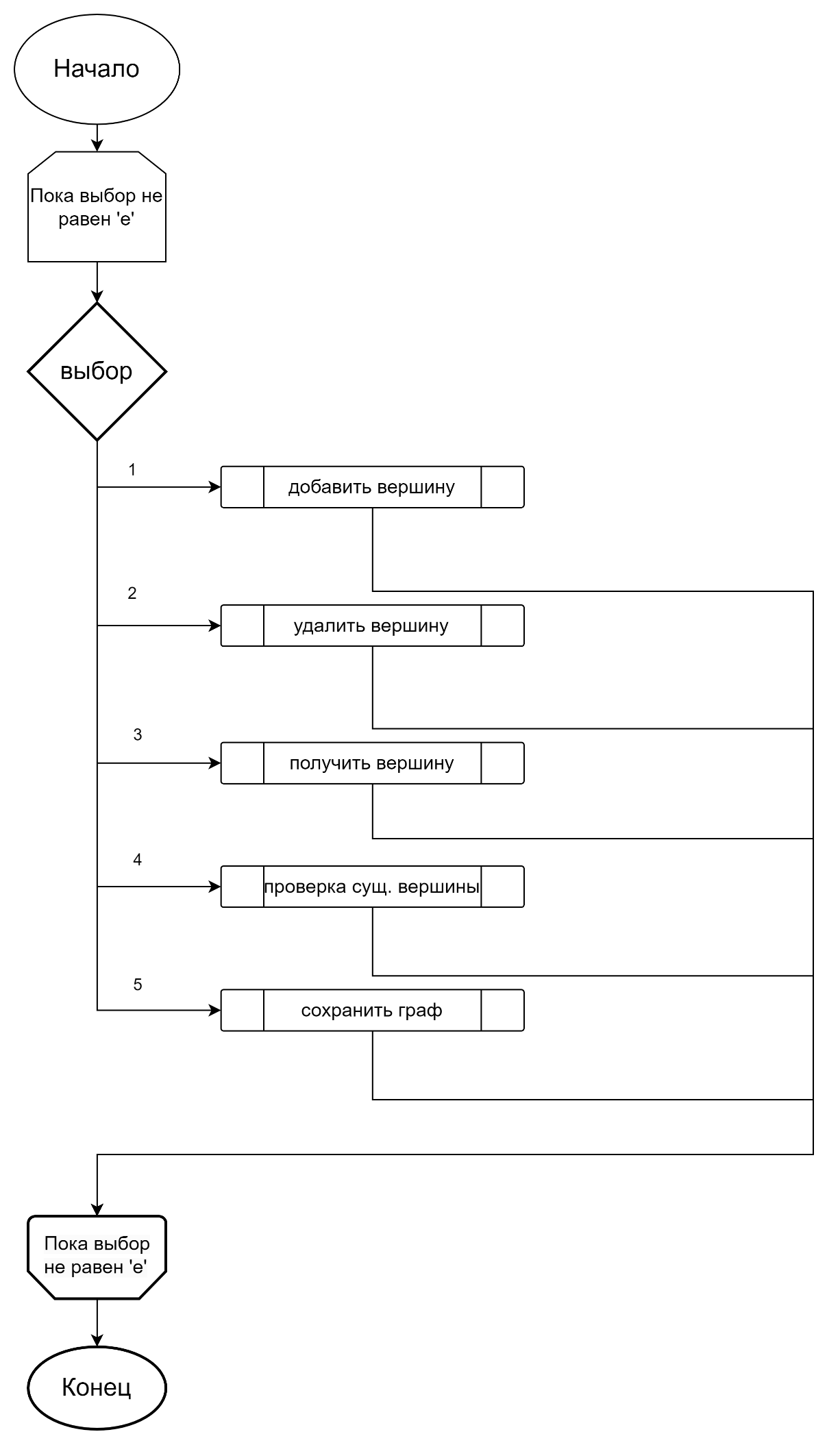


Рисунок 3 – Основное меню программы

Алгоритм поиска кратчайшего пути в графе основан на классическом алгоритме Дейкстры, с дополнительной проверкой типа дороги для велопоездок. Блок-схема алгоритма, изображенная на рисунке 1, ищет кратчайший путь между двумя вершинами. Его параметрами являются начальная вершина, конечная вершина и приемлемый тип дороги. Путем пошагового перебора всех вершин, аналогично обходу в ширину, весь граф постепенно исследуется. Просмотренные вершины больше не проходятся, так как им по мере их прохождения присваивается флаг просмотра.

Для реализации алгоритмов на графах используются сами графы, для которых предусмотрены функции добавления, редактирования и удаления вершин и ребер. Для визуализации результатов работы алгоритма используются сторонние решения, такие как Graphviz.

После сохранения текущего графа программа форматирует его в специальный файл input.dot для последующего отображения на экране с помощью заранее написанного скрипта для чтения данных из этого файла. В результате формируется изображение output.png, на котором представлен сформированный граф из текстового описания.



Рисунок 4 – Алгоритм поиска - начало

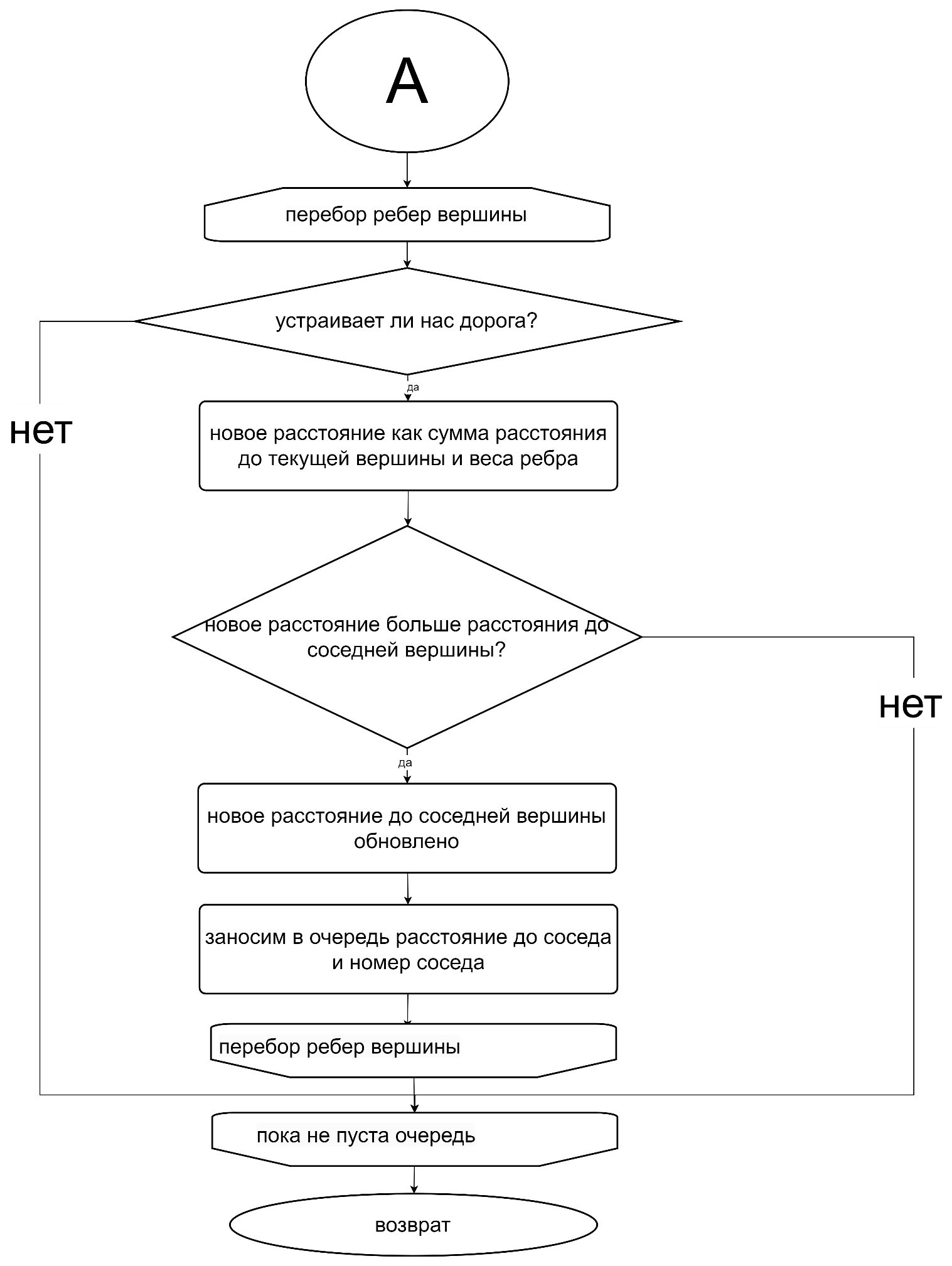


Рисунок 4.A – Алгоритм поиска - продолжение

Здесь представлен алгоритм вывода кратчайшего пути из вектора path, который формируется по мере прохождения по контейнеру distances.

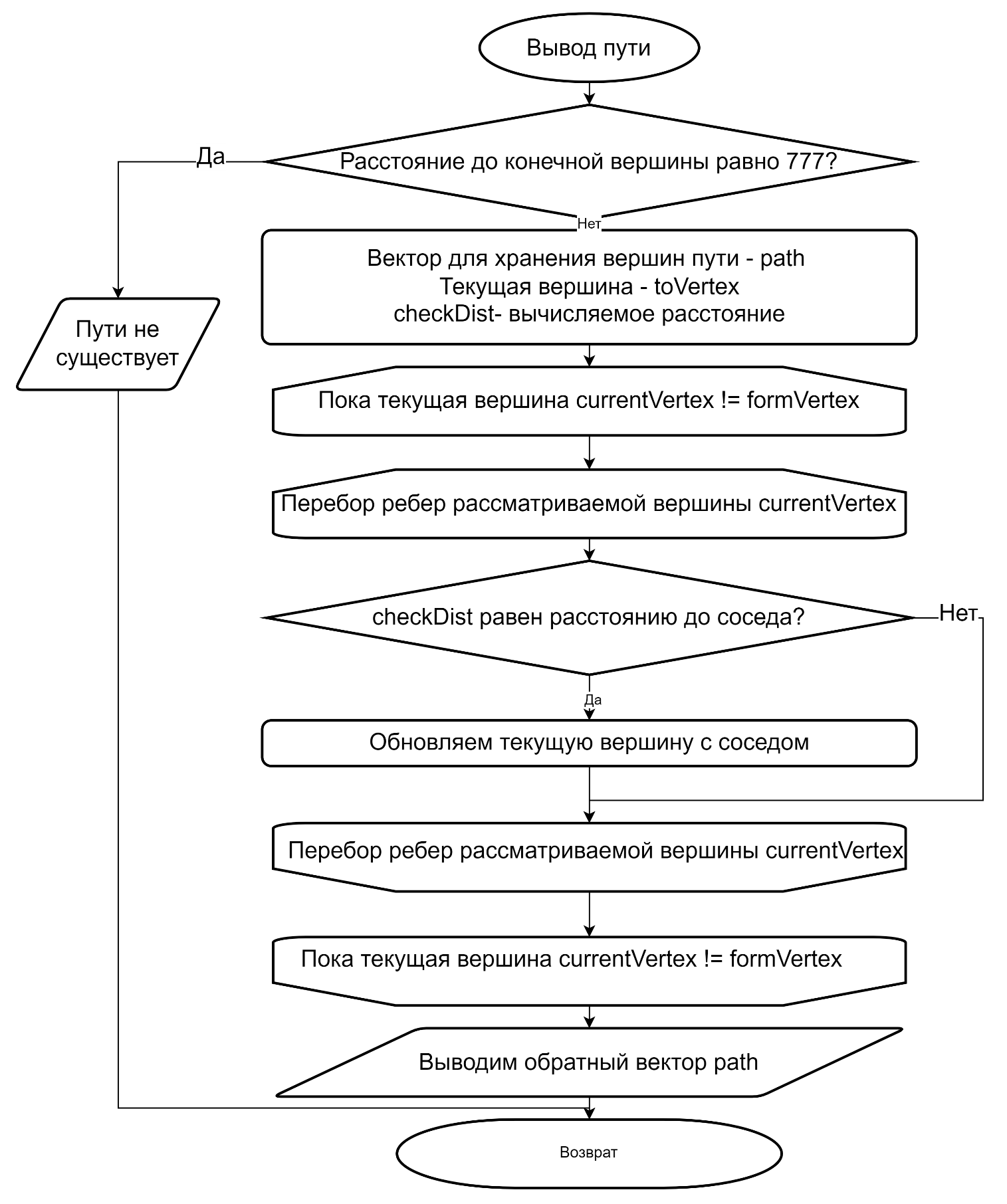


Рисунок 5 – Алгоритм вывода кратчайшего пути

# 3 Программная реализация структур данных

Для реализации алгоритма Дейкстры в графическом интерфейсе на языке программирования C++ были разработаны классы Graph, Vertex, Edge. Ниже приведена спецификация полей классов Graph, включающий основные поля и методы для выполнения алгоритма и визуализации графа, Vertex, Edge.

Таблица 1 – Поля класса Graph

|  |  |  |
| --- | --- | --- |
| Тип данных | Наименование поля | Функциональное назначение |
| vector<Vertex> | vertices | Основной массив с вершинами графа |
| string | pathStr | Хранение пути |

В таблице 2 приведена спецификация функций класса Graph.

Таблица 2 – Функции обработки структур данных

|  |  |  |  |
| --- | --- | --- | --- |
| Название функции | Функциональное назначение | Входные параметры | Возвращаемое значение |
| addVertex | Добавление новой вершины в граф | int id - идентификатор новой вершины | void – ничего не возвращает |
| addEdgeByID | Добавление нового ребра в граф | int sourceId - идентификатор начальной вершины  int destId - идентификатор конечной вершины  nt weight - вес ребра  int traffic - трафик ребра | void – ничего не возвращает |
| getVertexById | Получение вершины по идентификатору | int id - идентификатор вершины | Vertex - вершина графа |
| dijkstra | Алгоритм Дейкстры для поиска кратчайшего пути | int fromVertex - идентификатор начальной вершин  int acceptTraffic - ограничение по трафик | void – ничего не возвращает |
| checkIfEdgeExistById | Проверка существования ребра | int fromVertex – идентификатор начальной вершины, int toVertex – идентификатор конечной вершины | bool – ложь или истина |
| deleteVertexByID | Удаление вершины | int vid – идентификатор вершины | void – ничего не возвращает |
| saveGraphToFile | Показать граф | const string& filename | void |

Таблица 3 – Поля класса Edge

|  |  |  |
| --- | --- | --- |
| Тип данных | Наименование поля | Функциональное назначение |
| int | DestinationVertexId; | Вершина назначения |
| int | weight | Вес ребра |
| int | traffic | Тип дороги |

В таблице 4 приведена спецификация функций класса Edge.

Таблица 4 – Функции обработки структур данных

|  |  |  |  |
| --- | --- | --- | --- |
| Название функции | Функциональное назначение | Входные параметры | Возвращаемое значение |
| getDestinationVertexId | Получение вершины назначения | void | int |
| getWeight | Получение веса ребра | void | int |
| setWeight | Назначение веса | int w | void |
| getTraffic | Получение дороги | void | int |
| setTraffic | Назначение дороги | int t | void |

Таблица 5 – Поля класса Vertex

|  |  |  |
| --- | --- | --- |
| Тип данных | Наименование поля | Функциональное назначение |
| int | id; | Номер вершины направления |
| list<Edge> | edgeList | Вес ребра |

В таблице 6 приведена спецификация функций класса Vertex.

Таблица 6 – Функции обработки структур данных

|  |  |  |  |
| --- | --- | --- | --- |
| Название функции | Функциональное назначение | Входные параметры | Возвращаемое значение |
| getEdgeList | Получение списка ребер вершины | void | list<Edge> |
| getID | Получение номера вершины направления | void | int |
| setID | Назначение номера вершины направления | int id | void |

# +

# 4 Анализ эффективности алгоритма решения задачи

Инициализация алгоритма Дейкстры требует времени, пропорционального количеству вершин графа, то есть O(V), где V - количество вершин.

Очередь с приоритетом используется для пошагового рассмотрения вершин с наименьшим расстоянием. Каждая вершина добавляется и извлекается из очереди не более одного раза за O(log V) операций. Всего операций на добавление и извлечение вершин из очереди будет O(E \* log V), где E - количество ребер.

Циклы алгоритма выполняются для каждой вершины (O(V)) и каждого ребра (O(E)).

Таким образом, временная сложность алгоритма Дейкстры составляет:

 (1)

При упрощении (игнорировании множителей и слагаемых меньшего порядка) получаем:

 (2)

. Временная сложность алгоритма Дейкстры описывается как линейно-логарифмическая относительно количества вершин (V) и ребер (E) в графе. Это связано с тем, что наиболее времязатратная операция - добавление и извлечение вершин из очереди с приоритетом - выполняется за время O(log V), а выполнение циклов для каждой вершины (O(V)) и каждого ребра (O(E)) добавляет линейную сложность.

# 5 Тестирование программного средства

Так выглядит меню программы с главным функционалом (рисунок 6).

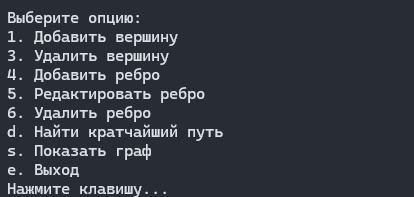


Рисунок 6 – Сообщение о нарушении условия выхода из лабиринта

Так выглядит построенный граф с помощью возможностей Graphviz и языка dot (рисунок 7).

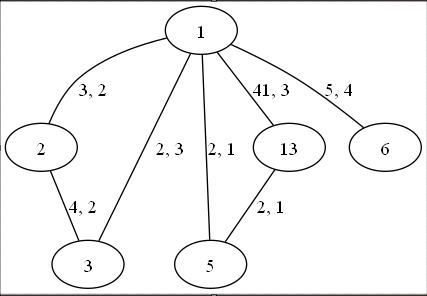


Рисунок 7 – Отрисованный граф

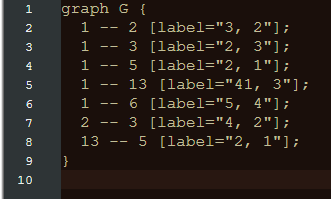


Рисунок 8 – Сформированный файл с расширением dot графа

Проверка корректности результата выполнения алгоритма поиска кратчайшего пути при выборе типа дороги - гравий. (рисунок 9).

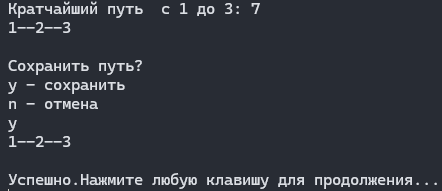


Рисунок 9 – Вывод программы после использования метода поиска пути

Таким образом, в результате тестирования была подтверждена корректность работы данного оконного приложения.

# Заключение

В ходе выполнения курсовой работы были применены алгоритмы Дейкстры для поиска кратчайших путей в графах, а также восстановления свойств бинарной кучи для эффективной обработки заданных массивов в контексте задачи о кратчайших путях. Для визуализации работы алгоритмов использовался инструментарий Graphviz, который автоматически визуализирует графы, представленные в виде DOT-описания, а также разнообразные программные решения для текстового и графического представления данных, применяемые при разработке программного обеспечения для визуализации структурированных данных.

Каждый этап алгоритма Дейкстры был подробно описан как в текстовой, так и в блок-схематической форме. Кроме того, были составлены спецификации для всех созданных функций, проведен анализ эффективности алгоритмов и выполнено тестирование для проверки правильности работы приложения в контексте поиска кратчайших путей в графе.

Таким образом, все инженерные задачи были успешно выполнены, цель курсовой работы достигнута, и применение алгоритмов Дейкстры в контексте работы с графами было успешно реализовано в разработанном приложении.Список использованных источников

1 Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. "Алгоритмы: построение и анализ". Москва: МИФИ, 2022. 1328 с. ISBN 978-5-907114-11-1, 978-5-8459-2016-4.

2 Тейлор, С. Р. "Язык программирования C++". Москва: Наука, 2021. 896 с. ISBN 978-5-4461-1085-9.

3 Макконнелл, С. "Совершенный код: Мастер-класс". Москва: Логос, 2021. 580 с. ISBN 978-5-905416-20-0.

4 Кнут, Д. "Искусство программирования. Том 1: Основные алгоритмы". Москва: Вильямс, 2020. 832 с. ISBN 978-5-907114-75-3.

5 Скиена, С. "Алгоритмы: введение в разработку и анализ". Москва: ДМК Пресс, 2019. 668 с. ISBN 978-5-94074-874-8.

6 Страуструп, Б. "Язык программирования C++". Москва: Вильямс, 2018. 1440 с. ISBN 978-5-8459-2222-9.

7 Хейнеманн, Г., Мараш, В. "Структуры данных и алгоритмы в C++". Москва: Вильямс, 2014. 632 с. ISBN 978-5-8459-1769-9.

8 Киган, Д., Мурер, М., Зайцев, Н. "STL. Поэтапное руководство". Москва: ДМК Пресс, 2015. 664 с. ISBN 978-5-94074-309-5.

9 Седжвик, Р. "Алгоритмы на C++: Справочник программиста". Москва: Диалектика, 2016. 320 с. ISBN 978-5-907114-65-4.

10 Калинников, Л. А., Хорошильцев, В. А. "Алгоритмы и структуры данных". Санкт-Петербург: БХВ-Петербург, 2017. 416 с. ISBN 978-5-9775-1833-1.

# Приложение А

# *(обязательное)*

# Программный код

#pragma once

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <queue>

#include <limits>

#include <iterator>

#include <list>

#include <conio.h>

#include<string>

#include <fstream>

#include <unordered\_set>

#include <algorithm>

#include <regex>

using namespace std;

class Edge {

private:

int DestinationVertexId;

int weight;

int traffic;

public:

Edge() {}

Edge(int destVID, int w, int t) {

DestinationVertexId = destVID;

weight = w;

traffic = t;

}

int getDestinationVertexId() {

return DestinationVertexId;

}

int getWeight() {

return weight;

}

void setWeight(int w) {

weight = w;

}

int getTraffic() {

return traffic;

}

void setTraffic(int t) {

traffic = t;

}

};

class Vertex {

private:

int id;

public:

list<Edge> edgeList;

Vertex() {

id = 0;

}

Vertex(int id) {

this->id = id;

}

int getID() {

return id;

}

void setID(int id) {

this->id = id;

}

list<Edge> getEdgeList() {

return edgeList;

}

};

class Graph {

public:

// список вершин

vector<Vertex> vertices;

// хранение пути

string pathStr;

int sp;

// добавить вершину

void addVertex(Vertex newVertex) {

if (checkIfVertexExistByID(newVertex.getID())) {

cout << "Вершина с таким номером уже существует.\n";

}

else {

vertices.push\_back(newVertex);

cout << "Новая вершина добавлена.\n";

}

}

// существует ли вершина?

bool checkIfVertexExistByID(int vid) {

for (int i = 0; i < vertices.size(); i++) {

if (vertices.at(i).getID() == vid) {

return true;

}

}

return false;

}

// получить вершину по номеру

Vertex getVertexByID(int v) {

Vertex temp;

for (int i = 0; i < vertices.size(); i++) {

temp = vertices.at(i); // по хорошему тут нужно обработать исключение

if (temp.getID() == v) {

return temp;

}

}

return temp;

}

// существует ли ребро?

bool checkIfEdgeExistById(int fromVertex, int toVertex) {

Vertex v = getVertexByID(fromVertex);

list <Edge> e;

e = v.getEdgeList();

for (auto it = e.begin(); it != e.end(); it++) {

if (it->getDestinationVertexId() == toVertex) {

return true;

}

}

return false;

}

// добавить ребро

void addEdgeByID(int fromVertex, int toVertex, int w, int t) {

for (int i = 0; i < vertices.size(); i++) {

if (vertices.at(i).getID() == fromVertex) {

Edge e(toVertex, w, t);

vertices.at(i).edgeList.push\_back(e);

// если граф неориетированный то нужно еще раз добавить от toVertex до fromVertex

}

else if (vertices.at(i).getID() == toVertex) {

Edge e(fromVertex, w, t);

vertices.at(i).edgeList.push\_back(e);

}

}

cout << "Ребро между вершинами " << getVertexByID(fromVertex).getID() << " и " << getVertexByID(toVertex).getID() << " добавлено.\n";

}

// удалить ребро

void deleteEdgeByID(int fromVertex, int toVertex) {

//if (checkIfEdgeExistById(fromVertex, toVertex)) {

for (int i = 0; i < vertices.size(); i++) { // проходим по вектору вершин

if (vertices.at(i).getID() == fromVertex) { // находим нужную вершину

for (auto it = vertices.at(i).edgeList.begin(); // проходим по списку ребёр этого узла

it != vertices.at(i).edgeList.end(); it++) { //

if (it->getDestinationVertexId() == toVertex) { // находим вторую нужную вершину

vertices.at(i).edgeList.erase(it); // удаляем ребро с первой вершины до второй

//cout << "Первое удаление\n";

break;

}

}

}

if (vertices.at(i).getID() == toVertex) { // проходим по вектору вершин

for (auto it = vertices.at(i).edgeList.begin(); // находим нужную вершину

it != vertices.at(i).edgeList.end(); it++) { // проходим по списку ребёр этого узла

if (it->getDestinationVertexId() == fromVertex) { //

vertices.at(i).edgeList.erase(it); // находим вторую нужную вершину

//cout << "Второе удаление\n"; // удаляем ребро со второй вершины до первой

break;

}

}

}

}

}

// обновить вершину

void updateEdgeByID(int fromVertex, int toVertex, int newWeight, int newTraffic) {

if (checkIfEdgeExistById(fromVertex, toVertex)) {

for (int i = 0; i < vertices.size(); i++) {

if (vertices.at(i).getID() == fromVertex) {

for (auto it = vertices.at(i).edgeList.begin();

it != vertices.at(i).edgeList.end(); it++) {

if (it->getDestinationVertexId() == toVertex) {

it->setWeight(newWeight);

it->setTraffic(newTraffic);

break;

}

}

}

else if (vertices.at(i).getID() == toVertex) {

for (auto it = vertices.at(i).edgeList.begin();

it != vertices.at(i).edgeList.end(); it++) {

if (it->getDestinationVertexId() == fromVertex) {

it->setWeight(newWeight);

it->setTraffic(newTraffic);

break;

}

}

}

}

}

else {

cout << "Ребро между " << getVertexByID(fromVertex).getID() << " и " <<

getVertexByID(toVertex).getID() << " не существует!\n";

}

}

// удалить вершину

void deleteVertexByID(int vid) {

int vIndex = 0;

for (int i = 0; i < vertices.size(); i++) {

if (vertices.at(i).getID() == vid) {

vIndex = i;

}

}

for (int i = 0; i < vertices.size(); i++) {

deleteEdgeByID(vertices.at(i).getID(), vid);

}

vertices.erase(vertices.begin() + vIndex);

cout << "Вершина удалена.\n";

}

struct PairHash {

template <class T1, class T2>

std::size\_t operator () (const std::pair<T1, T2>& p) const {

auto h1 = std::hash<T1>{}(p.first);

auto h2 = std::hash<T2>{}(p.second);

// Простое комбинирование хешей

return h1 ^ h2;

}

};

struct PairEqual {

template <class T1, class T2>

bool operator () (const std::pair<T1, T2>& lhs, const std::pair<T1, T2>& rhs) const {

return lhs.first == rhs.first && lhs.second == rhs.second;

}

};

void saveGraphToFile(const string& filename) {

ofstream file(filename);

if (!file.is\_open()) {

cerr << "Ошибка открытия файла для записи.\n";

return;

}

file << "graph G {\n";

std::unordered\_set<std::pair<int, int>, PairHash, PairEqual> addedEdges;//Множество для отслеживания уже добавленных рёбер

Vertex temp;

for (int i = 0; i < vertices.size(); i++) {

for (auto it = vertices.at(i).edgeList.begin(); it != vertices.at(i).edgeList.end(); it++) {

// Проверка, было ли уже добавлено такое ребро

// создаём ребро с двумя вершинами

pair<int, int> edge1 = { vertices.at(i).getID(),it->getDestinationVertexId() };

pair<int, int> edge2 = { it->getDestinationVertexId(), vertices.at(i).getID() };

// если нет этого ребра в unordered set то добавляем

if (addedEdges.find(edge1) == addedEdges.end() && addedEdges.find(edge2) == addedEdges.end()) {

file << " " << vertices.at(i).getID() << " -- " << it->getDestinationVertexId() << " [label=\"" << it->getWeight() << ", " << it->getTraffic() << "\"];\n";

addedEdges.insert(edge1); // Добавляем ребро в множество

\_getch();

}

}

}

file << "}\n";

}

void dijkstra(int fromVertex, int toVertex, list<int> accept\_roads) {

unordered\_map<int, int> distance;

unordered\_map<int, bool> visited;

for (Vertex vertex : vertices) {

distance[vertex.getID()] = 777;

visited[vertex.getID()] = false;

}

distance[fromVertex] = 0;

// очередь с приоритетами неубываюшая

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

pq.push({ 0, fromVertex }); // заносим в очередь начальную вершину

// пока очередь не пуста

while (!pq.empty()) {

// рассматриваемая вершина берет с помощью pq.top().second вершину из очереди ( с наименьшим расстоянием до этой вершины)

int currentVertex = pq.top().second;

// вытаскиваем эту вершину, так как мы её рассматриваем в переменной currentVertex

pq.pop();

// если вершина посещена то переходим к следующей итерации

if (visited[currentVertex]) {

continue;

}

// помечаем вершину как помеченую (если мы сюда дошли то вершина до этого не посещалась)

visited[currentVertex] = true;

// перебираем все ребра рассматриваемой вершины

for (Edge edge : getVertexByID(currentVertex).getEdgeList()) {

// соседняя вершина - конечная вершина ребра с наименьшим весом, которая есть у рассматриваемой вершины

int neighbor = edge.getDestinationVertexId();

// считаем новое расстояние учитывая эту вершину

int newDistance = distance[currentVertex] + edge.getWeight();

if (find(accept\_roads.begin(), accept\_roads.end(), edge.getTraffic()) == accept\_roads.end()) {

continue;

}

// ослабляем соседей

if (newDistance < distance[neighbor]) {

distance[neighbor] = newDistance;

pq.push({ distance[neighbor], neighbor });

}

}

}

sp = distance[toVertex];

cout << "Кратчайший путь с " << fromVertex << " до " << toVertex << ": " << distance[toVertex] << endl;

printShortestPath(fromVertex, toVertex, distance, pathStr);

cout << endl;

}

// печатает кратчайший путь

void printShortestPath(int fromVertex, int toVertex, unordered\_map<int, int>& distance, string& pathStr) {

if (distance[toVertex] == 777) {

cout << "Путь не найден";

return;

}

// вектор для хранения вершин кратчайшего пути

vector<int> path;

// текущая рассматриваемая вершина(начинаем с конца)

int currentVertex = toVertex;

Vertex tempVertex;

int checkDist;

int tempDist;

while (currentVertex != fromVertex) {

tempVertex = getVertexByID(currentVertex);

// перебираем ребра вершины (начинаем с конечной вершины)

for (Edge edge : tempVertex.getEdgeList()) {

// считаем расстояние между вершиной и предыдущей ей вершиной

checkDist = distance[currentVertex] - edge.getWeight();

// если расстояние до предыдущей совпадает с путем

if (checkDist == distance[edge.getDestinationVertexId()]) {

// то это нужная нам вершина, и мы с неё пришли!

// обновляем текущую вершину

currentVertex = edge.getDestinationVertexId();

// добавляем её в вектор пути вершин

path.push\_back(currentVertex);

// далее нет смысла смотреть другие вершины

break;

}

}

}

reverse(path.begin(), path.end());

pathStr.clear();

for (int vertex : path) {

pathStr += to\_string(vertex) + "--";

}

pathStr += to\_string(toVertex);

for (int vertex : path) {

cout << vertex;

cout << "--";

}

cout << toVertex;

cout << '\n';

}

};

void printMenu() {

cout << "Выберите опцию:" << endl;

cout << "1. Добавить вершину" << endl;

cout << "3. Удалить вершину" << endl;

cout << "4. Добавить ребро" << endl;

cout << "5. Редактировать ребро" << endl;

cout << "6. Удалить ребро" << endl;

cout << "d. Найти кратчайший путь" << endl;

cout << "s. Показать граф" << endl;

cout << "e. Выход" << endl;

}

void PrintSuccess() {

cout << "\nУспешно.Нажмите любую клавишу для продолжения...\n";

}

void PrintFailure() {

cout << "\nНеудачно. Нажмите любую клавишу для продолжения...\n";

}

void PrintWrong() {

cout << "\nНеверный выбор. Нажмите любую клавишу для продолжения...\n";

}

#include "Graph.h"

using namespace std;

int getId1() {

int id1;

cout << "Введите номер начальной вершины:\n";

while (!(std::cin >> id1)) {

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cout << "Ошибка: введите целое число.\n";

std::cout << "Введите номер начальной вершины:\n";

}

return id1;

}

int getId() {

int id;

cout << "Введите номер вершины:\n";

while (!(std::cin >> id)) {

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cout << "Ошибка: введите целое число.\n";

std::cout << "Введите номер вершины:\n";

}

return id;

}

int getId2() {

int id2;

cout << "Введите номер конечной вершины:\n";

while (!(std::cin >> id2)) {

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cout << "Ошибка: введите целое число.\n";

std::cout << "Введите номер конечной вершины:\n";

}

return id2;

}

int main() {

setlocale(LC\_ALL, "rus");

Graph graph;

string file = "input.dot";

bool check1, check2, check3;

int id;

int id1;

int id2;

int t;

int w;

Edge edge1;

Vertex v1;

list<int> selectedRoads;

int roadType;

char answer;

char choice;

do {

system("cls"); // cleaning screen

printMenu();

cout << "Нажмите клавишу...\n";

choice = \_getch();

switch (choice) {

case '1':

// Логика для добавления вершины

cout << "Вы выбрали: Добавить вершину" << endl;

id = getId();

v1.setID(id);

graph.addVertex(v1);

PrintSuccess();

\_getch();

break;

case '3':

// Логика для удаления вершины

cout << "Вы выбрали: Удалить вершину" << endl;

id = getId();

if (graph.checkIfVertexExistByID(id)) {

graph.deleteVertexByID(id);

PrintSuccess();

\_getch();

}

else {

cout << "Вершина с таким номером не существует.\n";

PrintFailure();

\_getch();

}

break;

case '4':

// Логика для добавления ребра

cout << "Вы выбрали: Добавить ребро" << endl;

id1 = getId1();

id2 = getId2();

if (!graph.checkIfEdgeExistById(id1, id2) && graph.checkIfVertexExistByID(id1) && graph.checkIfVertexExistByID(id2)) {

cout << "Введите вес ребра:\n"; cin >> w;

cout << "Введите тип дороги:\n"

<< "1 - велодорожка\n"

<< "2 - гравий\n"

<< "3 - грунт\n"

<< "4 - автомагистраль\n"; cin >> t;

graph.addEdgeByID(id1, id2, w, t);

PrintSuccess();

\_getch();

}

else {

cout << "Ребро между вершинами " << graph.getVertexByID(id1).getID() << " и " << graph.getVertexByID(id2).getID() << " уже существует или таких вершин нет.\n";

PrintFailure();

\_getch();

}

break;

case '5':

// Логика для редактирования ребра

cout << "Вы выбрали: Редактировать ребро" << endl;

id1 = getId1();

id2 = getId2();

if (graph.checkIfEdgeExistById(id1, id2) && graph.checkIfVertexExistByID(id1) && graph.checkIfVertexExistByID(id2)) {

cout << "Введите НОВЫЙ вес ребра:\n"; cin >> w;

cout << "Введите НОВЫЙ тип дороги:\n"

<< "1 - велодорожка\n"

<< "2 - гравий\n"

<< "3 - грунт\n"

<< "4 - автомагистраль\n"

<< "5 - назад\n"; cin >> t;

graph.updateEdgeByID(id1, id2, w, t);

PrintSuccess();

\_getch();

}

else {

cout << "Ребро между вершинами " << graph.getVertexByID(id1).getID() << " и " << graph.getVertexByID(id2).getID() << " не существует.\n";

PrintFailure();

\_getch();

}

break;

case '6':

// Логика для удаления ребра

cout << "Вы выбрали: Удалить ребро" << endl;

id1 = getId1();

id2 = getId2();

if (graph.checkIfEdgeExistById(id1, id2)) {

graph.deleteEdgeByID(id1, id2);

PrintSuccess();

\_getch();

}

else {

cout << "Ребро между вершинами " << graph.getVertexByID(id1).getID() << " и " << graph.getVertexByID(id2).getID() << " не существует.\n";

PrintFailure();

\_getch();

}

break;

case 'e':

cout << "Выход" << endl;

break;

case 's':

// Логика для показа из файла графа

cout << "Вы выбрали: Показать граф" << endl;

graph.saveGraphToFile(file);

system("showgraph.bat");

\_getch();

break;

case 'd':

// Логика для нахождения кратчайшего пути

cout << "Вы выбрали: Кратчайший путь" << endl;

id1 = getId1();

if (graph.checkIfVertexExistByID(id1)) {

id2 = getId2();

if (graph.checkIfVertexExistByID(id2)) {

cout << "Введите приемлемые дороги:\n"

<< "1 - велодорожка\n"

<< "2 - гравий\n"

<< "3 - грунт\n"

<< "4 - автомагистраль\n"

<< "5 - назад\n";

while (true) {

std::cin >> roadType;

if (roadType >= 1 && roadType <= 4) {

selectedRoads.push\_back(roadType);

}

else if (roadType == 5) {

break;

}

else {

std::cout << "Некорректный ввод. Попробуйте снова.\n";

break;

}

}

graph.dijkstra(id1, id2, selectedRoads);

if (graph.sp == 777) {

cout << "нет такого пути\n";

}

else {

cout << "Сохранить путь?\n"

<< "y - сохранить\n"

<< "n - отмена\n";

std::cin >> answer;

if (answer == 'y') {

cout << graph.pathStr << endl;

}

else {

cout << "Не сохраняем путь...\n";

}

}

PrintSuccess();

selectedRoads.clear();

\_getch();

}

else {

cout << "Вершина с таким номером не существует.\n";

PrintFailure();

\_getch();

}

}

else {

cout << "Вершина с таким номером не существует.\n";

PrintFailure();

\_getch();

}

break;

\_getch();

break;

default:

PrintWrong();

\_getch();

break;

}

} while (choice != 'e');

}