МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт Математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

02.03.03 Математическое обеспечение и администрирование информационных систем
(код и наименование направления подготовки / специальности)

WEB -дизайн и мультимедиа
(направленность (профиль)/специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Разработка программного обеспечения для решения задачи поиска кратчайшего пути в графе

Обучающийся А.Э. Александрова
(Инициалы Фамилия) (личная подпись)

Руководитель к.ф.-м.н. О.В. Лелонд
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант к.ф.н., доцент М.М. Бажутина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2022

Аннотация

Тема бакалаврской работы – «Разработка программного обеспечения для решения задачи поиска кратчайшего пути в графе».

Целью бакалаврской работы является создание ПО, использующего алгоритмы нахождения кратчайших путей взвешенного ориентированного графа.

Актуальность работы представлена в практической реализации методов поиска кратчайшего пути в графе и анализа готовых решений для поиска наиболее оптимального по соотношению затрачиваемого времени и памяти вычислительного устройства.

Бакалаврская работа состоит из введения, трёх глав, заключения, списка использованной литературы и одного приложения.

Во введении раскрывается актуальность исследования, конкретизируется цель работы и устанавливаются задачи для решения проблемы.

Первая глава работы посвящена описанию задачи, описанию и выбору конкретных методов решения задачи, а также обзору состояния проблемы в других исследованиях.

Во второй главе бакалаврской работы проводится выбор и сравнение алгоритмов поиска кратчайшего пути, а также практическая реализация выбранных алгоритмов в виде программного кода.

Третья глава работы описывает проведение вычислительного эксперимента по поиску кратчайшего пути графа и анализ его результатов.

В заключении работы подводится итог выполненных мероприятий и выделяется практическая и теоретическая значимость работы.

Бакалаврская работа представлена в объёме 46 страниц, содержит 18 рисунков, 2 таблицы, 6 листингов программного кода и 1 приложение.

Abstract

The title of the bachelor's work is "Development of software for solving the problem of finding the shortest path in a graph."

The purpose of the bachelor's work is to create software that uses algorithms for finding the shortest paths of a weighted directed graph.

The relevance of the work is presented in the practical implementation of methods for finding the shortest path in a graph and analyzing ready-made solutions to find the most optimal computing device in terms of the ratio of time spent and memory.

The bachelor's thesis consists of an introduction, three chapters, a conclusion, a list of references and one appendix.

The introduction reveals the relevance of the study, specifies the purpose of the work and sets the tasks for solving the problem.

The first chapter of the work is devoted to the description of the problem, the description and choice of specific methods for solving the problem, as well as an overview of the state of the problem in other studies.

In the second chapter of the bachelor's work, the selection and comparison of algorithms for finding the shortest path, as well as the practical implementation of the selected algorithms in the form of a program code, is carried out.

The third chapter of the work describes the implementation of a computational experiment to find the shortest path of the graph and the analysis of its results.

The conclusion summarizes the performed work and highlights the practical and theoretical significance of the work.

The volume of the bachelor's thesis is 46 pages, it also contains 18 figures, 2 tables, 6 code listings and 1 appendix.

Оглавление

Введение

Поиск кратчайшего пути от одной вершины графа до другой на сегодняшний момент является одной из наиболее востребованных математических задач, и имеет обширную зону практического применения: транспортная логистика, начиная от масштабных грузовых перевозок, заканчивая персональными автомобильными навигаторами; системы автопилотов воздушных и морских судов; размещение товаров на складе; сетевые коммуникации; социологические связи между группами людей и многое другое. Проблемы кратчайшего пути важны при изучении сетей. Во-первых, они очень распространены на практике; часто нужно найти лучший маршрут для отправки чего-либо из одного места в другое. Во-вторых, они обеспечивают основу для ряда ключевых идей предметного материала; рассматривая проблемы пути, можно заложить основу для большого количества других, более сложных сетевых моделей.

Если раньше поиск кратчайшего пути в графе мог занимать огромное количество времени в случае масштабных и запутанных графов, то с появлением первых ЭВМ эта задача стала более тривиальной. Вычислительные мощности современных устройств позволяют буквально «на лету» вычислять сложные пути и корректировать их в зависимости от внешних обстоятельств. Таким образом, актуальность бакалаврской работы заключается в реализации практического решения актуальной проблемы. Такое решение представляет практический интерес, поскольку может быть использовано для дальнейшего решения проблемы поиска кратчайшего пути в графе, а также теоретический, поскольку позволяет найти наиболее оптимальный алгоритм для решения каждой задачи.

Объект исследования бакалаврской работы – теория графов.

Предмет исследования – проблема поиска кратчайшего пути графа и конкретные алгоритмы решения данной проблемы.

Целью ВКР является разработка программного обеспечения для решения задачи поиска кратчайшего пути в графе, а также проведение анализа

алгоритмов, позволяющего выявить наиболее оптимальный алгоритм решения поставленной проблемы.

Для достижения цели ВКР необходимо выполнить следующие задачи:

– провести анализ предметной области и определить итоговую формулировку решаемой задачи;

– провести аналитический обзор состояния проблемы в работах других исследователей для конкретизации используемых алгоритмов;

– выбрать алгоритмы, используемые для решения поставленной задачи;

– реализовать выбранные алгоритмы в виде программного кода;

– протестировать полученные программы и выбрать наиболее оптимальный алгоритм для решения поставленной задачи.

Структурно ВКР состоит из введения, трёх глав, заключения и приложения.

В первой главе рассматривается теоретическая основа поставленной задачи, её описание и решение в работах других авторов, а также математическое описание задачи для дальнейшего поиска решения.

Вторая глава ВКР посвящена алгоритмам решения задачи поиска кратчайшего пути в графе. В ней подробным образом описываются алгоритмы, их достоинства и недостатки в контексте решаемой задачи, а затем описывается реализация выбранных алгоритмов в виде программ.

В третьей главе бакалаврской работы полученные программы проходят тестирование, их работа профилируется, полученные данные подвергаются анализу и на основании данных о работе каждого алгоритма принимается решение о возможности использования его для решения искомой задачи. Кроме этого, выявляется наиболее оптимальный алгоритм для решения задачи поиска кратчайшего пути в графе.

Глава 1 Анализ исследуемой задачи

1.1 Описание и постановка задачи поиска кратчайшего пути в графе

Понятие графа служит для математического изучения таких ситуаций, когда имеются две совокупности объектов, причем объекты второй группы играют роль связок, соединяющих пары предметов первой группы друг с другом (конкретно речь может идти, например, об отдельных деталях электрической схемы и соединяющих проводниках; об учреждениях и курьерах, доставляющих бумаги из одного в другое; о людях и отношениях любви, дружбы или родства между ними; о целых числах и отношении делимости; и т. д.). Для одной и той же пары предметов допускается одновременное наличие нескольких связей, среди которых могут быть и односторонние, и двусторонние; возможны связи, соединяющие предмет с самим собой. В математическом смысле граф представляет собой совокупность двух множеств [1]: вершин графа, и парных связей между ними – рёбер графа. Как правило, граф обозначают диаграммой: вершины точками или кругами, рёбра графа – линиями. Существуют различные виды графов [12], рассмотрим некоторые из них:

– Простой граф (рисунок 1) $\mathrm{G}(\mathrm{V},\mathrm{E}) = \langle \mathrm{V},\mathrm{E} \rangle,\ \mathrm{V} \neq \emptyset,\ \mathrm{E} \subseteq \mathrm{V} \times \mathrm{V},$ $\{v, v\} \notin E,\ v \in V$, где v – элемент множества V вершин графа G, E – множество рёбер (неупорядоченных пар элементов множества V) графа G.

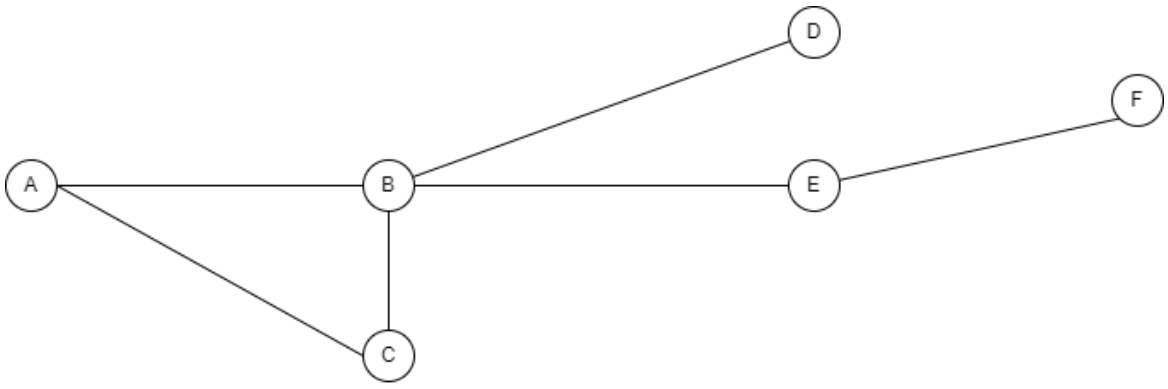

Рисунок 1 – Простой граф

– Псевдограф (рисунок 2) $\mathrm{G(V,E)} = \langle \mathrm{V,E} \rangle,\ \mathrm{V} \neq \emptyset,\ \mathrm{E} \subseteq \mathrm{V} \times \mathrm{V}$. В отличие от простого графа, в множестве рёбер псевдографа разрешены элементы $\{v, v\} \in E$, то есть рёбра, у которых совпадают концевые вершины. Такие рёбра называются петлями.

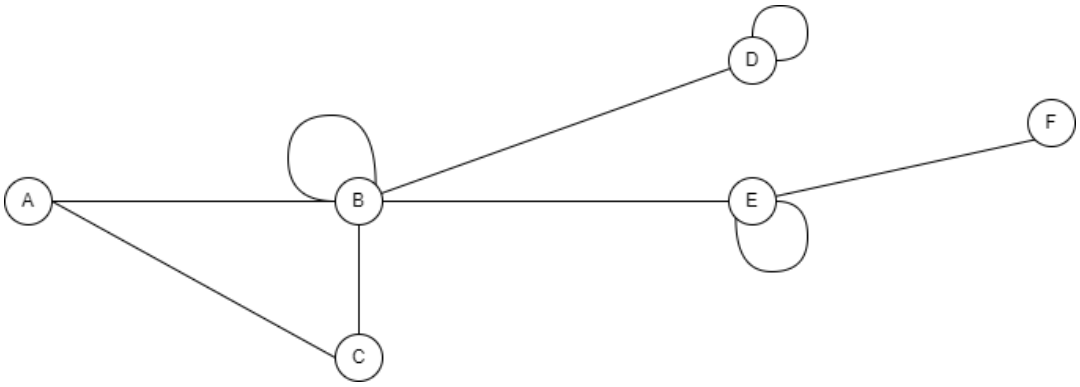

Рисунок 2 – Псевдограф. Вершины B, D, E имеют рёбра-"петли"

– Мультиграф (рисунок 3) $\mathrm{G(V,\mathbf{E})} = \langle \mathrm{V,\mathbf{E}} \rangle,\ \mathrm{V} \neq \emptyset,\ \mathbf{E} \subseteq \mathrm{V} \times \mathrm{V},\ \{v, v\} \notin \boldsymbol{E},\ v \in V$. От простого графа отличается тем, что в мультиграфе $\mathbf{E}$ – мультимножество, то есть содержит одинаковые элементы. На диаграмме графа такие рёбра изображаются дополнительными линиями между вершинами.

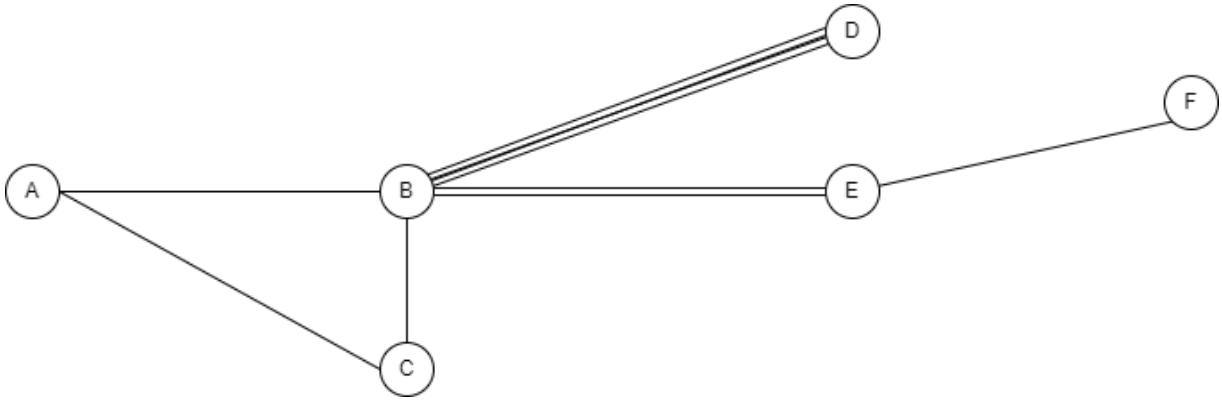

Рисунок 3 – Мультиграф. Пары вершин BD и BE имеют более одного ребра

– Ориентированный граф (рисунок 4) $\mathrm{G}(\mathrm{V},\mathrm{E}) = \langle \mathrm{V},\mathrm{E}\rangle,\ \mathrm{V} \neq \emptyset,\ \mathrm{E} \subseteq \mathrm{V} \times \mathrm{V},\ \langle\{v_1, v_2\}\rangle \in E,\ v \in V$. В ориентированном графе рёбра представлены дугами, связывающими две вершины графа таким образом, что каждая дуга e представима в виде упорядоченной пары вершин(u, v), где u – начало дуги, а v– конец дуги. Также говорят, что дуга $u \rightarrow v$ ведёт от вершины u к вершине v, и такие вершины являются смежными.

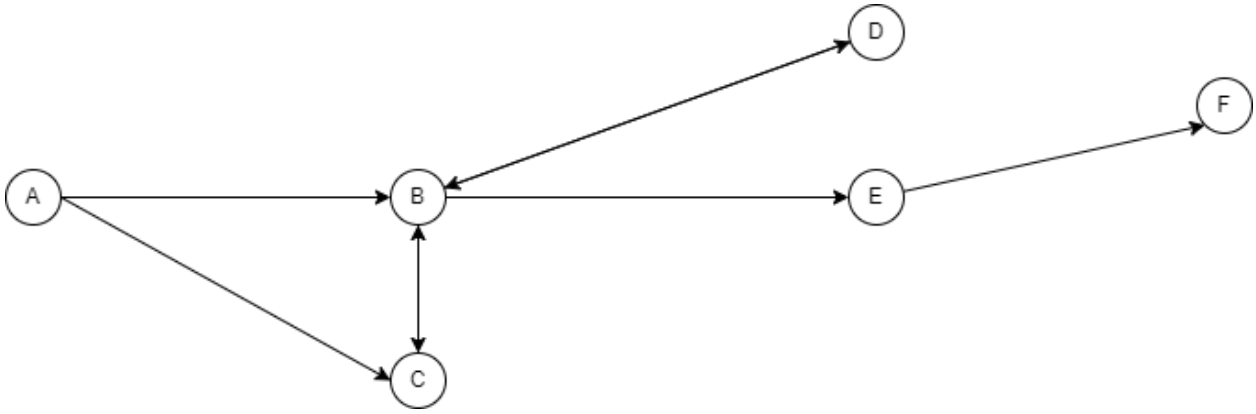

Рисунок 4 – Орграф. Направление дуг отображается стрелочками

– Смешанный граф (рисунок 5) $\mathrm{G}(\mathrm{V},\mathrm{E},\mathrm{U}) = \langle \mathrm{V},\mathrm{E},\mathrm{U}\rangle,\ \mathrm{V} \neq \emptyset,\ \mathrm{E} \subseteq \mathrm{V} \times \mathrm{V},\ \langle\{v_1, v_2\}\rangle \in E, \{v_3, v_4\} \in U,\ v \in V$. Смешанный граф представляет собой комбинацию ориентированного и неориентированного графов. В нём присутствуют как множество дуг E, так и множество рёбер U.

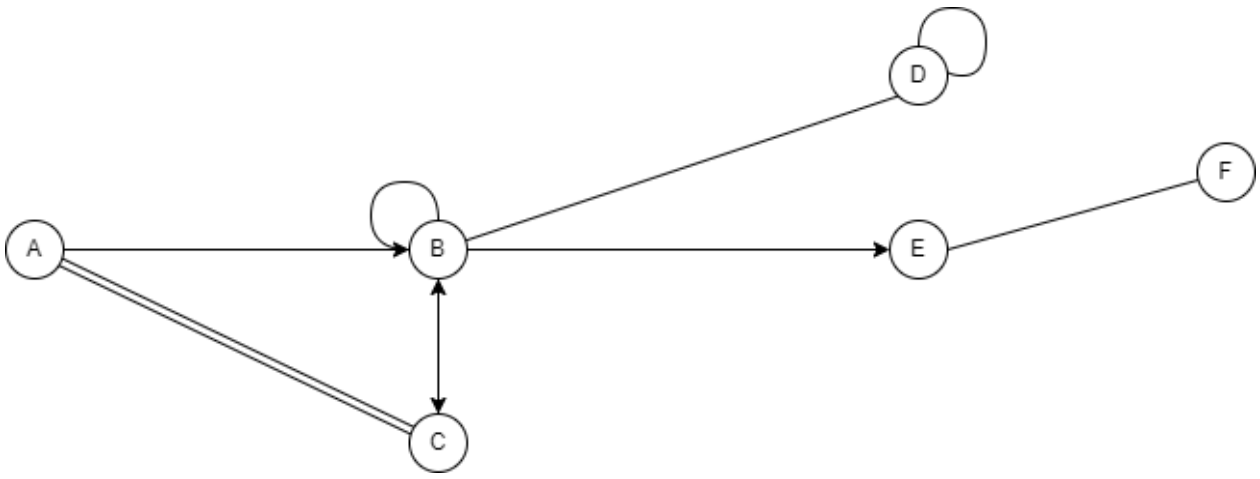

Рисунок 5 – Смешанный граф

Также граф обладает некоторыми другими свойствами, которые влияют на решение задачи поиска пути графа. Граф называется связным, если для любых двух вершин u, v существует путь из u в v (рисунок 6).

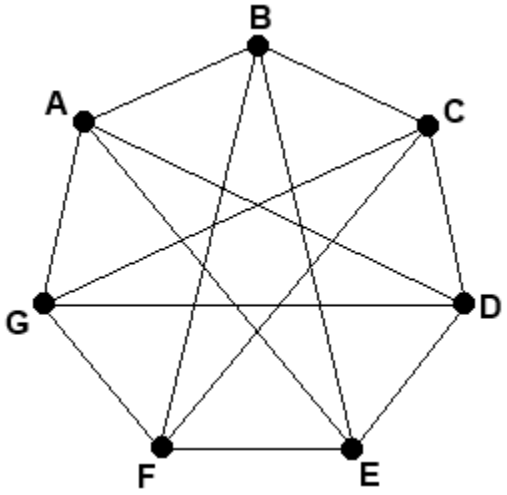

Рисунок 6 – Связный граф

Граф называется полным, если любые две его вершины соединены ребром [3]. Граф называется взвешенным, если каждому элементу множества рёбер графа соответствует некоторое число, которое называют весом ребра [4].

На практике наибольший интерес представляет решение задачи поиска кратчайшего пути в связных взвешенных ориентированных графах [9]. Именно такими графами являются, например, автомобильные дороги. Составление навигатором оптимального пути из точки А до точки Б можно представить в математическом смысле именно как решение задачи нахождения кратчайшего пути в ориентированном взвешенном графе. Суть решения данной задачи сводится к тому, чтобы из заданной вершины графа u найти кратчайший путь до заданной вершины v. Эффективность решения задачи определяется по скорости работы алгоритма, который выявляет путь с наименьшим общим весом (суммой веса всех рёбер выбранного пути от вершины u до вершины v).

Есть и другие задачи, которые можно сформулировать как задачи о кратчайшем пути. К ним относятся проблема рюкзака, планирование производства и составление графика работы персонала на 24-часовой график.

Определим итоговую формулировку задачи. Дан планарный ориентированный взвешенный граф $\mathrm{G}(\mathrm{V}, \mathrm{E}, w)$, где V – вершины графа, E – множество упорядоченных пар вершин $u, v \in V$ (дуги графа), w – функция, определяющая числовые значения $L \in \mathbb{Q},\ L \geq 0$ дуг графа E (вес дуг графа). Стоит отметить, что вес рёбер в задаче не может быть отрицательным, поскольку это значительно затрудняет поиск кратчайших путей. Вес пути от вершины u до вершины v является суммой весов всех ребер пути. Кратчайшим путем от вершины u до вершины v является такой путь, при котором значение веса пути будет минимальным. Если существует несколько таких путей с минимальным весом, то все они являются кратчайшими. Для удобства дальнейших вычислений обозначим минимальный вес пути как σ_G.

1.2 Выбор и обоснование принципа и методов решения задачи

Существует несколько наиболее эффективных и популярных алгоритмов для решения данной задачи, в зависимости от её постановки, рассмотрим некоторые из них:

– Алгоритм Дейкстры, разработан нидерландским исследователем Эдсгером Дейкстрой в 1959 году. Используется для того, чтобы найти кратчайшие пути от заданной вершины до всех остальных вершин. Не работает в случае взвешенного графа с рёбрами отрицательного веса [8].

– Алгоритм Беллмана-Форда. Был предложен двумя учёными, работавшими независимо друг от друга: Ричардом Беллманом и Лестером Фордом. Алгоритм представляет собой усовершенствованную версию алгоритма Дейкстры и допускает наличие рёбер с отрицательным весом.

– Волновой алгоритм (также известен как алгоритм Ли). Основывается на алгоритме поиска в ширину и работает для планарных

графов. Включает в себя три этапа: инициализацию, распространение волны и восстановление пути. В момент инициализации алгоритма строится обрабатываемое дискретное рабочее поле, каждой ячейке которого присваивается вес, атрибут проходимости, а также определяются стартовая и финишная ячейки. Затем при распространении волны в каждую проходимую ячейку записывается число шагов (либо вес в случае взвешенного графа) от стартовой до тех пор, пока «волна» не дойдёт до финишной ячейки. После этого в обратном порядке восстанавливается кратчайший путь до стартовой ячейки.

– Алгоритм Флойда-Уоршелла. В основе алгоритма лежит идея о том, что кратчайший путь из вершины u в v состоит из других кратчайших путей к промежуточным вершинам, в чём напоминает работу жадного алгоритма, но учитывает все пары вершин, а не наиболее очевидные варианты. Имеет высокую вычислительную сложность $O(n^3)$. При применении его к решению задач на значительно разветвленном графе, он потребует значительных затрат со стороны ресурсов вычислительной машины [10].

– Алгоритм Косарайю. Данный алгоритм опирается на метод поиска в глубину. Для начала выполняется определенный поиск в глубину для обратного графа, или другими словами, на обращении исходного графа (то есть графа, который может получиться при инвертировании ребер исходного, начального графа. Далее идет вычисление вектора обратного порядка обхода [8]. На втором шаге происходит поиск в глубину на исходном графе, причем надо заметить, что вершины берутся в том порядке, который является обратным тому, который, в свою очередь, получился в следствие нумерации вершин при обратном проходе при первом запуске поиска в глубину (по полученному вектору обхода). Когда выполнятся метод поиска в глубину, то используются непосещенные вершины, которые имеют максимальный номер. Если выполнить весь алгоритм, то в конечном итоге получится лес, у которого деревья будут представлять сильные компоненты связности графа [8].

– Алгоритм A*. Впервые описан в 1968 году группой учёных и является расширенным при помощи эвристики алгоритмом Дейкстры. Работает он аналогичным образом, однако имеет приоритеты для исследования наиболее оптимальных узлов, в чём похож на жадный алгоритм, однако в A* учитывается путь до вершины из стартовой точки, а не из последнего узла.

– Алгоритм поиска в ширину. Метод слепого поиска, при котором производится полный обход графа, подсчёт величин всех рёбер и конечный поиск минимального значения. При поиске в ширину алгоритм систематически обходит все рёбра известных вершин для открытия новых вершин, достижимых из исходной до тех пор, пока не перестанет обнаруживать новые вершины графа.

– Жадный алгоритм поиска кратчайшего пути. Принцип жадного алгоритма заключается в принятии оптимальных решений на каждом этапе для получения оптимального конечного решения. То есть, алгоритм каждый раз будет выбирать вершины с минимальным весом на пути к заданной конечной точки и игнорировать менее оптимальные варианты на каждом узле. Такой алгоритм не всегда способен дать глобально оптимальное решение [10].

Перечисленные алгоритмы можно разделить на две группы – алгоритмы перебора и эвристические алгоритмы. Перебор позволяет гарантированно найти кратчайший путь между двумя вершинами (если такой существует), но скорость работы алгоритма сильно зависит от количества вершин. Так, например, скорость работы алгоритма Дейкстры составляет $O(V^2)$. Эвристические алгоритмы используют методы приблизительного определения следующей вершины (эвристические приближения) и дают большую скорость работы алгоритма ценой возможных неоптимальных значений.

1.3 Аналитический обзор состояния проблемы в литературе

Проблема выявления кратчайшего пути в графе – одна из базовых проблем теории графов, в явном виде впервые упоминается в трудах Денеша Кёнинга в 1936 году [15]. Нет точных данных о том, что эта проблема рассматривалась ранее в математических исследованиях в явном виде. Большинство ранних трудов были направлены на составление минимального остовного дерева в графе [10]. Подробно проблема поиска кратчайшего пути была изучена Джозефом Краскалом в 1956 году, и затем исследования были продолжены Эдсгером Дейкстрой, который в 1959 году предложил собственный алгоритм для нахождения кратчайших путей от исходной вершины до всех остальных [14].

В СССР проблема поиска кратчайшего пути рассматривалась в переводах иностранных научных трудов [11], и к тому времени были созданы уже несколько оптимальных алгоритмов для решения задачи, поэтому кардинально нового наши соотечественники в решение проблемы не привнесли. Больший интерес в исследованиях представляли другие задачи комбинаторной оптимизации теории графов, например, задача коммивояжёра. Значительный вклад в исследование теории графов в СССР внёс А.А. Зыков. Дальнейшие исследования теории графов опирались на базис, заданный иностранными первопроходцами, однако, в последние годы наблюдаются тенденции в увеличении количества российских исследований [12].

Современные подходы к решению задачи поиска кратчайшего пути предполагают использование других эвристических методов – нейросетей, искусственного интеллекта и многоэтапных алгоритмов. Актуальность такого направления исследования состоит в том, что приближенные методы способны только дать одно из решений, которое может существенно отличаться от оптимального. При этом сложно сделать заключение, насколько это решение близко к точному, так как нет вычислительных мощностей, чтобы получить его. Единственное что возможно – попытаться улучшить полученные значения аргументов. Наблюдать результат уточнения можно по

уменьшению (или увеличению) значения целевой функции при применении различных операций.

Одним из перспективных современных алгоритмов решения задач теории графов, таких как задача коммивояжёра или поиска кратчайшего пути является т.н. «муравьиный алгоритм» [13]. Этот алгоритм был разработан на основе наблюдения за движением муравьёв, создающих пути для перемещения пищи к колонии. Муравьи изначально перемещаются в случайных направлениях, пока не найдут источник пищи. Затем они возвращаются в колонию, оставляя за собой след из феромонов. Другие муравьи, обнаружив такой след, с большей вероятностью пойдут по нему, чем по другому пути. Со временем путь из феромонов начнёт испаряться, и чем больше времени требуется для прохождения пути, тем быстрее испарится тропа. На более коротком пути плотность феромонов будет оставаться выше за счёт постоянного поддержания другими муравьями, не исключая, впрочем, возможность поиска новых путей, однако если новый путь окажется более длинным, он не получит достаточного количества феромонов для дальнейшей поддержки. Таким образом, за счёт испарения феромонов, муравьи стремятся к поиску локального оптимального решения. Данный алгоритм используется как сам по себе в различных вариациях, так и в качестве модели для обучения нейронных сетей [13].

Таким образом, проблема поиска кратчайшего пути до сих пор имеет актуальность. Учёные до сих пор находят различные новые решения задачи, которые имеют практическое применение в логистике, картографии, биологии, авиации и многих других областях науки. Существует множество алгоритмов для разных формулировок задачи, с разной эффективностью в зависимости от исходных данных графа [5][6].

Выводы по Главе 1

В данной главе были проанализированы виды графов, даны их математические определения и определены особенности. Была

сформулирована задача для дальнейшего решения. Кроме того, были рассмотрены виды алгоритмов, применяемые для решения задач поиска кратчайшего пути.

Были рассмотрены алгоритмы, использование которых возможно для решения поставленной задачи [7]: алгоритм Дейкстры, алгоритм Беллмана-Форда, алгоритм Ли (волновой алгоритм), алгоритм Флойда-Уоршелла, алгоритм Косарайю, алгоритм A*, поиск в ширину, жадный алгоритм поиска кратчайшего пути.

Также в данной главе была рассмотрена проблема поиска кратчайшего пути графа в работах отечественных и зарубежных исследователей. Были описаны работы как основополагателей теории графов, так и современных учёных.

Глава 2 Разработка и реализация алгоритмов решения задачи

2.1 Используемые методы и алгоритмы решения задачи

В данной работе мы используем несколько алгоритмов для определения наилучшего решения по соотношению скорости выполнения и точности результата. Рассмотрим подробнее работу этих алгоритмов.

Волновой алгоритм. Как ранее было обозначено, работа волнового алгоритма основывается на поиске в ширину и состоит из трёх основных этапов: инициализация, распространение волны, восстановление пути. Данный алгоритм отлично подходит для решения задач нахождения кратчайшего пути на планарном графе [17]. Рассмотрим, как пошагово работает волновой алгоритм (рисунок 7).

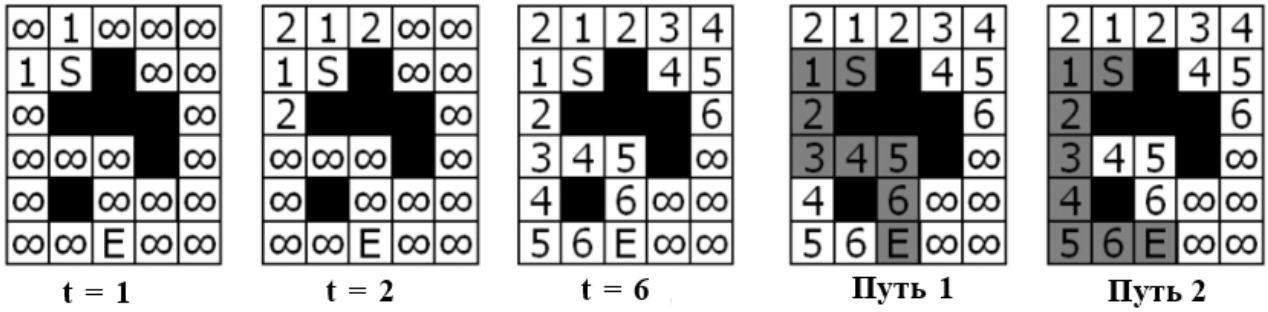

Рисунок 7 – Пошаговая работа волнового алгоритма

Шаг 1. Инициализация алгоритма. Стартовой точке присваивается значение i:=0.

Шаг 2. Распространение волны. Всем вершинам, доступным из исходной, присваивается значение i:=i+1. В случае взвешенного графа, значение i вершины будет составлять i + (значение веса ребра). Волна повторно распространяется до тех пор, пока не будет достигнута целевая вершина.

Шаг 3. Восстановление пути. Восстановление начинается из целевой вершины. Алгоритм ищет вершины со значением меньшим, чем у текущей

вершины, и добавляет эту вершину в «путь» до стартовой точки до тех пор, пока не доберётся до исходной вершины. Затем алгоритм отбрасывает все пути, не содержащие исходную вершину, оставляя только корректные пути, и из них выбирает кратчайший.

В виде псевдокода работа алгоритма выглядит следующим образом:

Листинг 1. Псевдокод работы алгоритма Ли

```
Отметить исходную вершину 0
i:=0
ЦИКЛ
ПОКА (целевая вершина не достигнута) И (есть смежные вершины)
    ДЛЯ каждой вершины n, отмеченной числом i
    отметить смежные неотмеченные вершины числом i+1
    i:=i+1
КОНЕЦ ЦИКЛА
ЕСЛИ целевая вершина достигнута
ТО
    переход в целевую вершину
    ЦИКЛ
        ПОКА (исходная вершина не достигнута)
        выбрать вершину со значением на 1 меньше текущего
        переход к выбранной вершине
        добавить вершину к пути
    КОНЕЦ ЦИКЛА
    ВОЗВРАТ (путь)
```

Если не задавать алгоритму конечную точку, он отрабатывается как алгоритм поиска в ширину, рассчитывая расстояние и путь от исходной вершины до всех остальных вершин графа.

Жадный алгоритм. Максимально простой и интуитивно понятный алгоритм. Чаще всего применяется в другой похожей на проблему кратчайшего пути задаче – задаче о рюкзаке. Жадный алгоритм выбирает на каждом этапе наиболее оптимальный вариант, предполагая, что конечный результат также окажется оптимальным. То есть в случае графа алгоритм всегда выбирает вершину с наименьшим весом ребра до тех пор, пока не достигнет целевой вершины. Из-за особенностей работы алгоритма, мы можем получить различные пути, поменяв местами исходную и целевую вершины (рисунок 8).

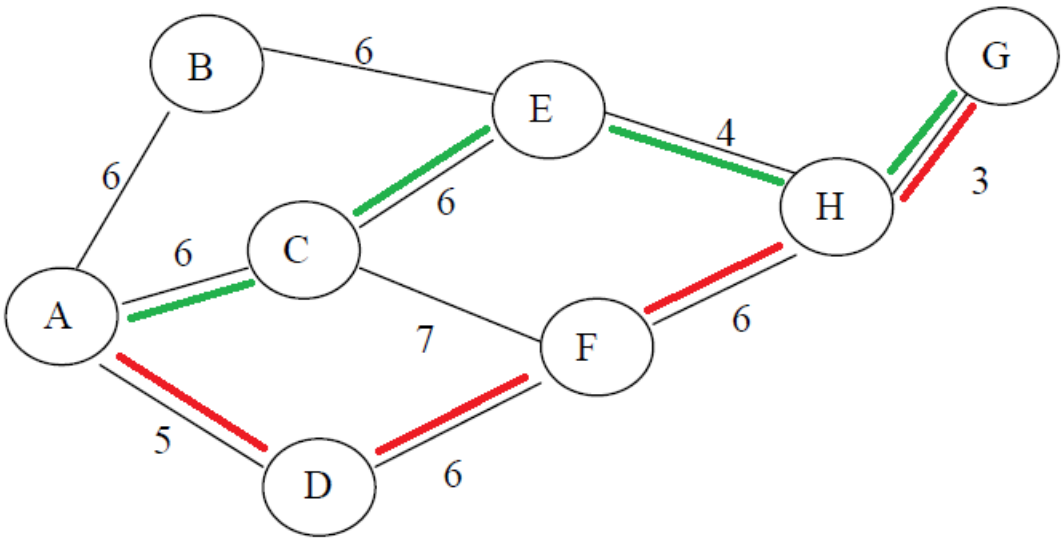

Рисунок 8 – Работа жадного алгоритма на примере простого взвешенного графа

Исходная вершина на представленном графе – A, конечная вершина – G. Красным цветом обозначен путь из A в G, определённый жадным алгоритмом. Общий вес пути в таком случае составит 20 условных единиц. Зелёным цветом обозначен путь из G в A. Вес пути в таком случае составит 19 условных единиц. Этот пример ярко показывает недостаток жадного алгоритма перед

другими алгоритмами поиска кратчайшего пути. Но жадный алгоритм имеет неоспоримое преимущество в виде скорости работы и простоты реализации. Псевдокод работы жадного алгоритма приведён ниже.

Листинг 2. Псевдокод работы жадного алгоритма.

```
Инициализация стартовой вершины
ЦИКЛ
    ПОКА (целевая вершина не найдена)
        ЕСЛИ (есть смежные вершины)
        выбрать вершину с минимальным весом ребра до неё
        перейти в выбранную вершину
        ИНАЧЕ
        пометить текущую вершину как неоптимальную
        вернуться в предыдущую вершину
КОНЕЦ ЦИКЛА
```

Иногда жадный алгоритм действительно может приводить к оптимальным результатам [18], но, если вопрос времени работы алгоритма не имеет критического влияния, предпочтительно использование другого алгоритма. Существуют также вариации жадного алгоритма, модифицированного при помощи генетических алгоритмов и нейросетей, а также способами кластеризации.

Алгоритм Дейкстры. Классический алгоритм поиска кратчайшего пути, позволяющий найти кратчайшие пути от искомой вершины графа до всех остальных [14][15]. Принцип работы алгоритма следующий (рисунок 9):

Рисунок 9 – Блок-схема упрощённой реализации алгоритма Дейкстры

Шаг 1. Инициализируется исходная вершина. Ей присваивается значение 0, поскольку расстояние до неё равно нулю. Всем другим вершинам графа присваивается недостижимо большое число, поскольку расстояние до них неизвестно. Создаётся список посещённых и непосещённых вершин (рисунок 10).

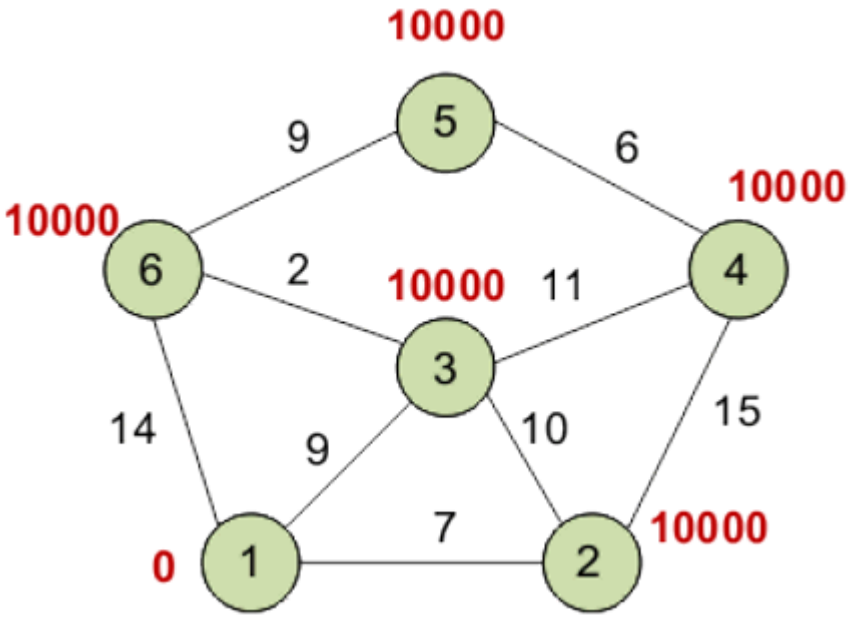

Рисунок 10 – Инициализация алгоритма Дейкстры

Шаг 2. Алгоритм выявляет смежные с исходной вершины и расстояние до них. Вершинам присваиваются значения, соответствующие расстоянию до них из исходной вершины, если они меньше уже присвоенного значения (рисунок 11).

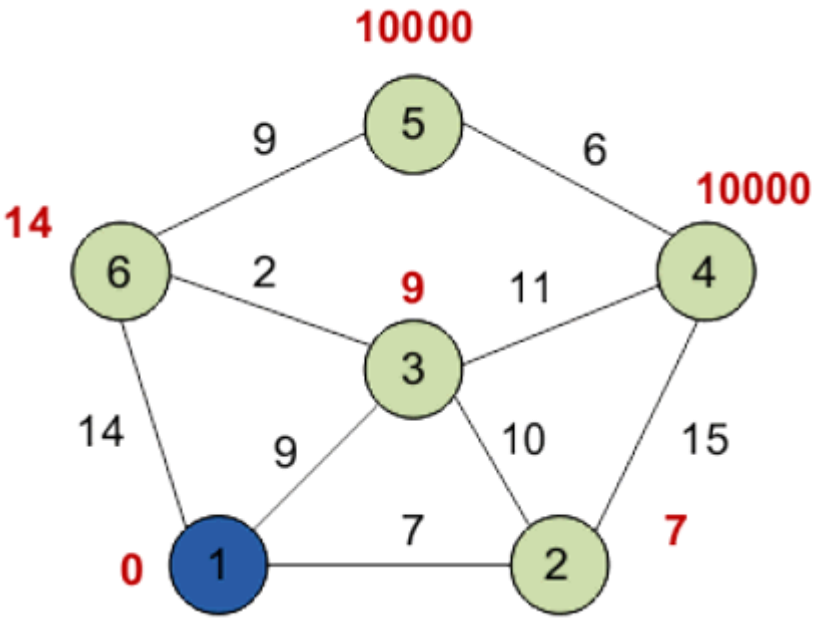

Рисунок 11 – Первый шаг алгоритма Дейкстры

Поскольку изначально данным вершинам присвоено недостижимое число, метки вершин обновляются на новые. Исходная вершина заносится в список посещённых, затем алгоритм переходит в следующую вершину.

Обход графа начинается с вершины, имеющей минимальное присвоенное значение (рисунок 12).

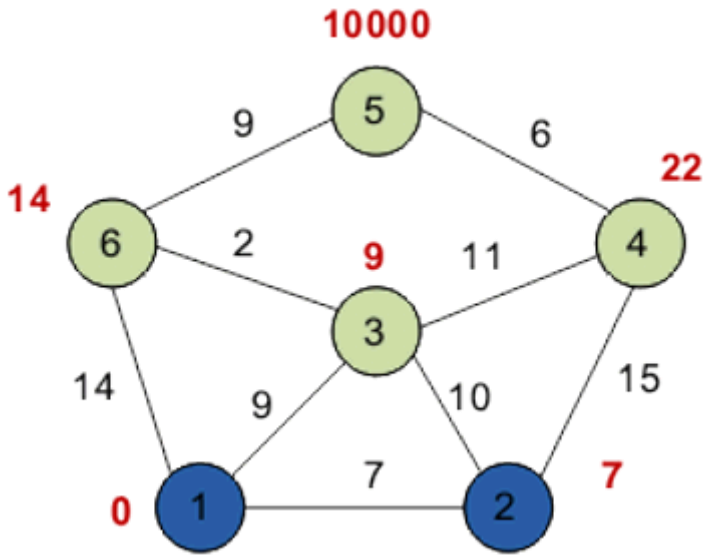

Рисунок 12 – Первый этап работы алгоритма

Шаг 3. Повторяется предыдущий шаг – алгоритм выявляет новые смежные вершины и рассчитывает расстояние до них, но уже с учётом ранее пройденного расстояния из исходной вершины в текущую.

Если такое расстояние меньше отметки, присвоенной вершине, присваивается новое минимальное значение. После оценки всех смежных вершин, текущая вершина заносится в список посещённых.

Далее алгоритм переходит в следующую вершину с минимальной отметкой, до тех пор, пока не будет пройден весь граф (рисунок 13).

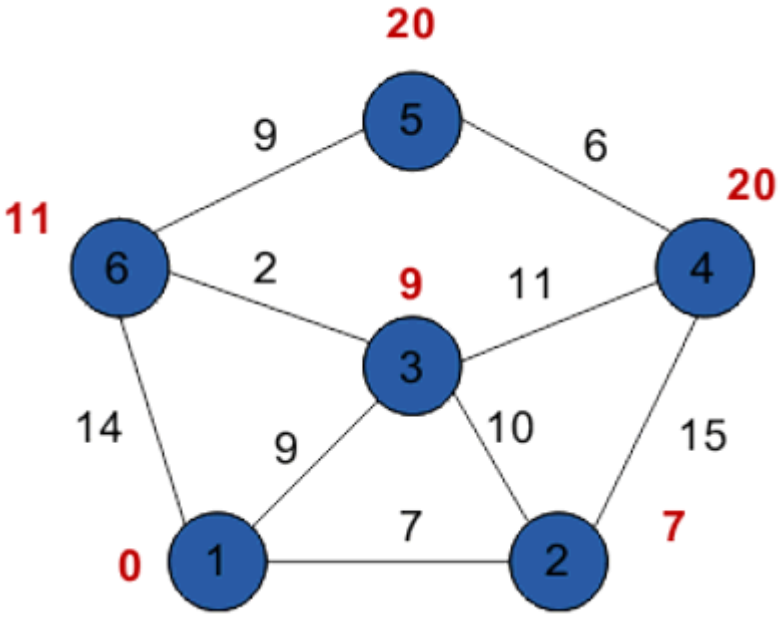

Рисунок 13 – Полный обход графа алгоритмом Дейкстры

Шаг 4. После прохода всего графа мы получим значение, присвоенное итоговой вершине, которое будет обозначать минимальный вес пути до неё. Чтобы восстановить путь, алгоритм начинает обход с конечной вершины, вычитая из её метки значение веса рёбер до соседних вершин. Если значение после вычитания совпало с меткой вершины, к которой ведёт ребро, алгоритм добавляет эту вершину в список пути и осуществляет переход в неё, где продолжает вычитать значение рёбер уже из метки новой вершины. В итоге алгоритм дойдёт до исходной вершины и вернёт список вершин кратчайшего пути.

Алгоритм Дейкстры может применяться только для графов с рёбрами неотрицательного веса, в иных случаях рекомендуется использовать различные его модификации, либо альтернативные алгоритмы.

Алгоритм A*. Данный алгоритм является расширенной версией алгоритма Дейкстры. Но в отличие от линейного алгоритма Дейкстры, A* использует эвристические методы для предположения оптимального пути [19]. Алгоритм A* обладает двумя важнейшими свойствами для алгоритмов поиска: оптимальность и полнота. Оптимальность алгоритма гарантирует, что среди всех результатов будет найден наиболее оптимальный, а полнота

говорит о том, что если путь решения существует, то с помощью данного алгоритма он гарантированно будет найден. Для работы алгоритма используется следующая формула: $f(v) = g(v) + h(v)$, где $f(v)$ – стоимость перехода к вершине v, $g(v)$ – вес пути до вершины v, $h(v)$ – эвристическое приближение веса пути от вершины v до конечной вершины. Таким образом, алгоритм будет выбирать вершины с минимальным значением $f(v)$ до тех пор, пока либо не дойдёт до конечной вершины, либо не зайдёт в «тупик». Визуальное представление работы алгоритма на планарном графе представлено на рисунке 14.

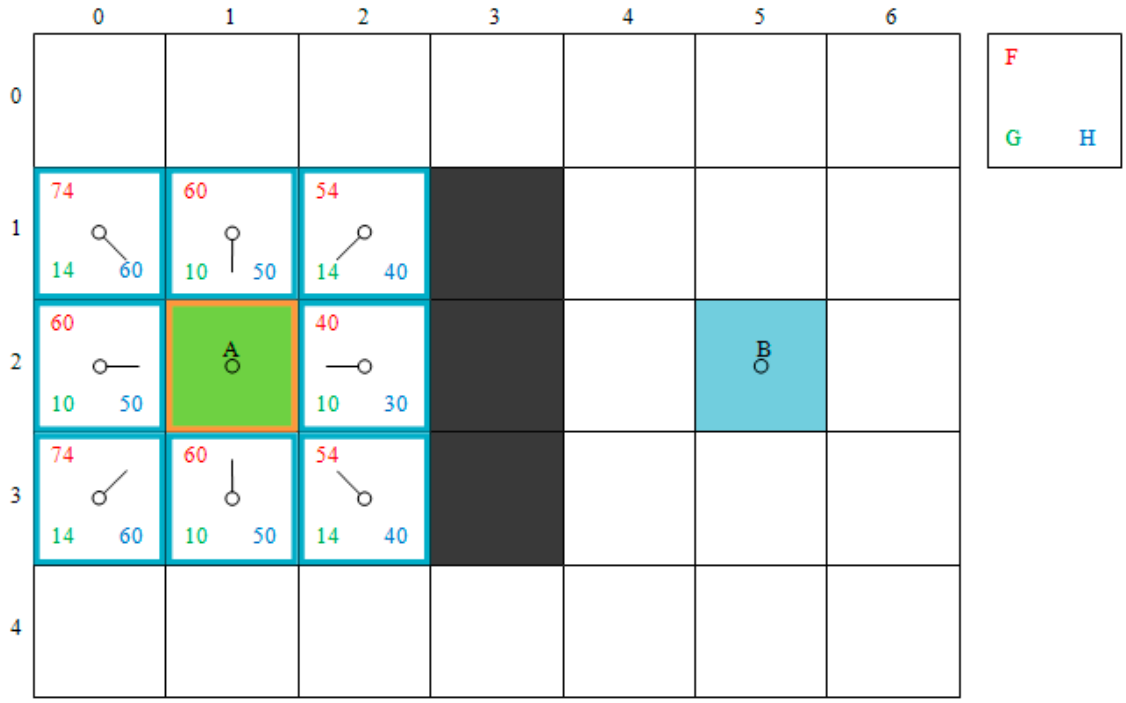

Рисунок 14 – Пример работы алгоритма A*

В данном примере эвристическое приближение основывается на минимальном количестве клеток от искомой точки до конечной, игнорируя движение по диагонали (вес которого больше, чем движение по прямой) и препятствия. Таким образом, алгоритм сначала выберет клетку с индексом [2;2], затем упрётся в препятствие и начнёт работу заново, выбрав соседние клетки. Следующим шагом алгоритм вновь вычисляет показатель f каждой

ячейки и находит дальнейший путь, пока не окажется в конечной точке (рисунок 15). Достигнув её, алгоритм выводит обратный путь до стартовой точки, опираясь на ранее пройденные ячейки [19].

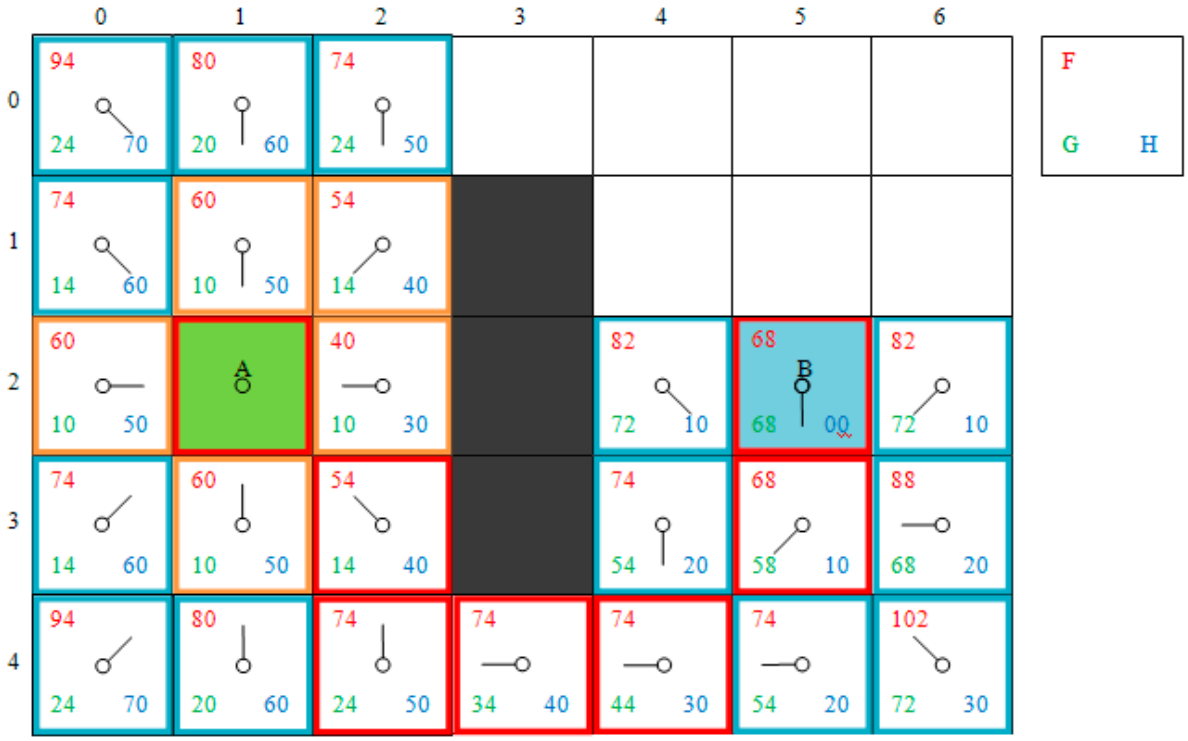

Рисунок 15 – Итог работы алгоритма A*

Кратчайший путь до конечной точки изображён на схеме красным цветом. Скорость выполнения алгоритма A* зависит от эвристики. Сложность выполнения алгоритма полиноминальна, когда эвристика алгоритма соответствует условию $|h(x) - h^*(x)| \leq O(\log h^*(x))$, где $h^*(x)$ – оптимальная эвристика, а $h(x)$ – фактическая эвристика. Таким образом, ошибка ожиданий должна быть меньше логарифма оптимальной эвристики.

Блок-схема работы алгоритма A* представлена на рисунке 16.

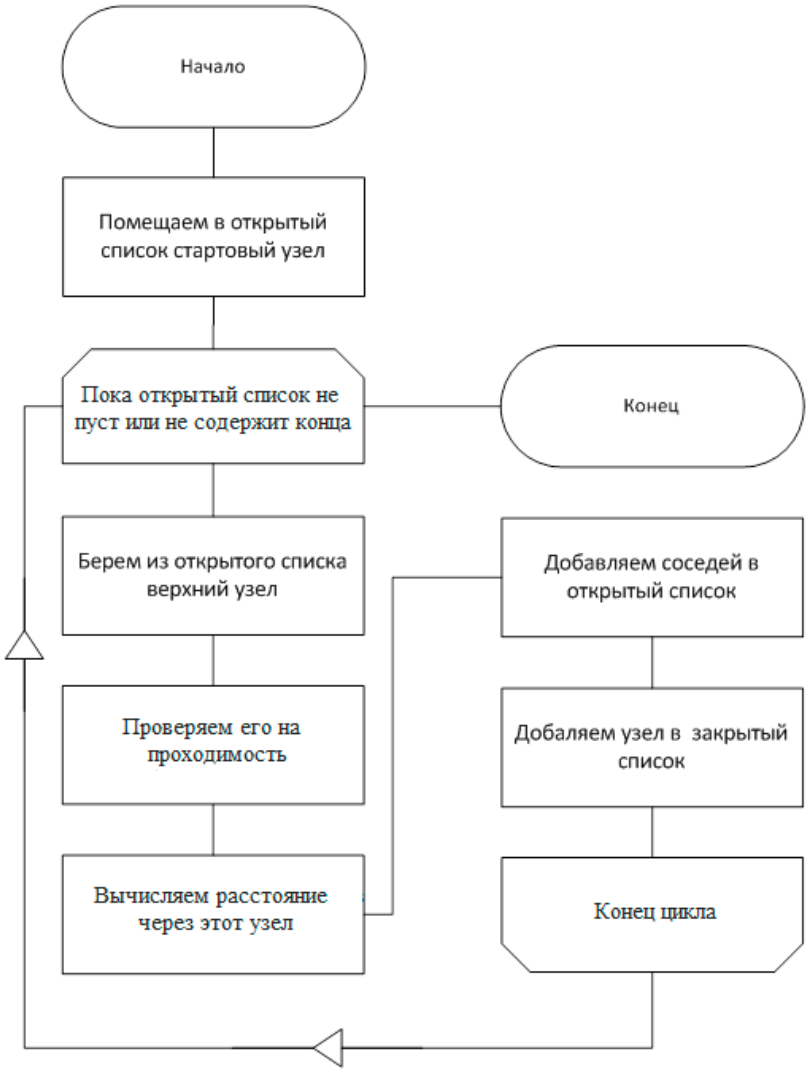

Рисунок 16 – Блок-схема работы алгоритма A*

2.2 Сравнение выбранных алгоритмов для решения поставленной задачи

Для выбора потенциально наиболее оптимального алгоритма в решении поставленной задачи необходимо провести сравнительный анализ описанных алгоритмов (таблица 1) по критериям его сложности, имеющихся ограничений алгоритма, а также преимуществ и недостатков каждого отдельно взятого алгоритма. Параметр временной сложности O в таблице указывается на основании стандартных, не модернизированных алгоритмов. В данном параметре переменная n – количество вершин графа, переменная m – количество рёбер графа.

Таблица 1 – Сравнительный обзор алгоритмов решения задачи кратчайшего пути

Название	Сложность	Ограничения	Преимущества	Недостатки
Алгоритм Ли	$O(n \times m)$	Работает на дискретном рабочем поле	Обладает полнотой Обладает оптимальностью Прост в реализации	Затраты времени на реализацию поиска существенно растут при увеличении размера графа Высокая требовательность к ресурсам
Жадный алгоритм	$O(n^3)$	Нет	Прост в реализации Обладает полнотой	Не обладает оптимальностью
Алгоритм Дейкстры	$O(n^2+m)$	Не работает в случае наличия ребер отрицательного веса	Обладает полнотой Обладает оптимальностью Небольшая сложность для полиноминального алгоритма	Проходит весь граф для поиска оптимального результата, что отражается на общем времени работы

Продолжение Таблицы 1

Название	Сложность	Ограничения	Преимущества	Недостатки
A*	-	Работает на дискретном рабочем поле	Обладает полнотой Обладает оптимальностью Не требует полного прохождения графа Использует эвристические приближения для оптимизации пути	Потребность в большом количестве памяти для хранения всех промежуточных величин При большой эвристической ошибке скорость выполнения алгоритма уступает альтернативам

Таким образом, в теории все представленные алгоритмы способны решить поставленную задачу. Эффективность каждого конкретного алгоритма можно определить только на практике, для чего необходимо реализовать все алгоритмы в виде программного кода.

2.3 Программная реализация выбранных алгоритмов

Определившись с набором алгоритмов для тестирования, мы можем перейти к их программной реализации. Необходимо выбрать, на базе какого языка программирования будет выполнена реализация. Рассмотрим некоторые варианты:

– C/C++. Для решения математических задач это один из самых популярных вариантов: согласно индексу популярности языков программирования TIOBE, язык C занимает первую строчку рейтинга, а C++ - четвёртую. Преимуществами Си являются возможность работы напрямую с

динамической памятью системы, а также общая скорость работы выполняемого алгоритма. Минусами выбора Си и его расширенной версии являются сложный синтаксис, мешающий пониманию работы кода с первого взгляда, а также проблемы с безопасностью в работе с указателями и массивами, поскольку при разработке на С можно достаточно легко выйти за границы доступной памяти.

– Java. Благодаря управляемой памяти, Java прощает больше ошибок при работе с кодом неопытному программисту, что увеличивает порог вхождения. Основное преимущество Java – кроссплатформенность. Один и тот же код, написанный на Java, работает в десктопных программах, мобильных приложениях и на веб-сайтах. Недостатком языка является удручающая производительность на фоне более производительных конкурентов.

– Python. Основной критерий при выборе Python – его простота. Язык подкупает своим интуитивным синтаксисом и способом написания кода. Благодаря этому фактору, Python является популярным не только среди программистов, но и в кругах других учёных: биологов, физиков, географов. Простота изучения Python и его гибкость позволяют автоматизировать многие статистические расчёты и визуализировать их результаты [2]. Недостатками Python являются жёсткая привязка к системным библиотекам и требовательность к доступной памяти.

– C#. Является неотъемлемой частью платформы .NET, которая принадлежит Microsoft, поэтому очень сильно, как и Python, зависит от наличия компонентов в системе. По синтаксису язык похож на С, ответвлением которого, по сути, и является, но область применения C# несколько отличается. Главное отличие от С – это, конечно, объектно-ориентированный подход, в отличие от процедурного С. Недостатками языка является статическая типизация, неудобная для реализации небольших проектов «на скорую руку», а также требовательность к аппаратным характеристикам системы.

Для реализации алгоритмов был выбран язык программирования Python, по причине его универсальности и имеющегося ранее опыта работы с указанным языком программирования. Это интерпретируемый, объектно-ориентированный и высокоуровневый язык программирования. Python называется интерпретируемым языком, поскольку его исходный код компилируется в байт-код, который затем интерпретируется. Python обычно компилирует код Python в байт-код перед его интерпретацией [2]. Он поддерживает динамическую типизацию и динамическую привязку [20].

В таких языках, как Java, C и C++, вы не можете инициализировать строковое значение переменной int, и в таких случаях программа не будет компилироваться [11]. Python не знает тип переменной до тех пор, пока код не будет выполнен. Для многих основное преимущество языка Python заключается в удобочитаемости программного кода, который в большинстве случаев более компактен, чем на других языках программирования [2]. Python имеет простой синтаксис, который улучшает читаемость и снижает затраты на обслуживание кода. Код выглядит понятно и коротко [20].

Главная особенность Python в контексте поставленной задачи – автоматическое управление памятью, что позволит отслеживать затраты на выполнение каждого алгоритма.

Реализация ключевого метода волнового алгоритма на языке Python имеет следующий вид:

Листинг 3. Функция работы волнового алгоритма

```
def gen_lee(start, size, travelable):
    neighbor_offsets = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    score = 0
    path_map = [[None for _ in xrange(size)] for _ in xrange(size)]
    node_list = [start]
    path_map[start[0]][start[1]] = 0
    for node in node_list:
```

```
        score = path_map[node[0]][node[1]]
        for neighbor_offset in neighbor_offsets:
            neighbor_x = node[0] + neighbor_offset[0]
            neighbor_y = node[1] + neighbor_offset[1]
            if neighbor_x < 0 or \
              neighbor_y < 0 or \
              neighbor_x >= size or \
              neighbor_y >= size:
                continue
            if not travelable[neighbor_x][neighbor_y]:
                continue
            if path_map[neighbor_x][neighbor_y] is None:
                node_list.append((neighbor_x, neighbor_y))
                path_map[neighbor_x][neighbor_y] = score + 1
    return path_map
```

Данная функция на вход получает некоторый массив, а затем совершает обход согласно алгоритму, описанному в пункте 2.1

Реализация метода жадного алгоритма:

Листинг 4. Функция работы жадного алгоритма

```
def best_first_search(source, target, vertices):
    visited = [0] * vertices
    pq = PriorityQueue()
    pq.put((0, source))
    print("Path: ")
    while not pq.empty():
        u = pq.get()[1]
        print(u, end=" ")
        if u == target:
```

```
            break
        for v, c in graph[u]:
            if not visited[v]:
                visited[v] = True
                pq.put((c, v))
    print()
```

Реализация алгоритма Дейкстры:

Листинг 5. Алгоритм Дейкстры

```
def dijkstra(graph, starting_vertex, destination_vertex):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[starting_vertex] = 0
    pq = [(0, starting_vertex)]
    while len(pq) > 0:
        current_distance, current_vertex = heapq.heappop(pq)
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
    return round(distances[destination_vertex], 2)
def build_graph(edges, weights, e):
    graph = edges
    for i in range(e):
        graph[i].append(weights[i])
      return graph
```

Реализация алгоритма A*:

*Листинг 6. Алгоритм A**

```
def astar(start, end):
    path = {}
    distance = {}
    q = priorityQueue()
    h = makeheuristicdict()
    q.push(start, 0)
    distance[start] = 0
    path[start] = None
    expandedList = []
    while (q.isEmpty() == False):
        current = q.pop()
        expandedList.append(current)
        if (current == end):
            break
        for new in graph[current]:
            g_cost = distance[current] + int(new.distance)
            if (new.point not in distance or g_cost < distance[new.point]):
            distance[new.point] = g_cost
            f_cost = g_cost + heuristic(new.point, h)
            q.push(new.point, f_cost)
            path[new.point] = current
```

Алгоритмы реализованы в программном виде, теперь необходимо выяснить, какие из них лучше подходят для решения поставленной задачи.

Выводы по Главе 2

Во второй главе бакалаврской работы были подробно рассмотрены выбранные алгоритмы поиска кратчайшего пути, изложен пошаговый принцип их работы, составлены листинги псевдокода и блок-схемы работы.

Были рассмотрены волновой алгоритм, жадный алгоритм поиска кратчайшего пути, алгоритм Дейкстры и алгоритм A*. Был проведён сравнительный анализ алгоритмов, выявивший их ограничения работы, достоинства и недостатки. В результате анализа было принято решение реализовывать все указанные алгоритмы, поскольку все они подходят для решения задачи.

В качестве языка реализации программных алгоритмов был выбран Python благодаря своей универсальности и простоте.

Наиболее важная задача была выполнена – все выбранные алгоритмы были реализованы в виде программного кода на языке Python для дальнейшей работы с ними.

Глава 3 Проведение вычислительных экспериментов

3.1 Выбор методики проведения эксперимента. Подготовка оборудования и программного обеспечения

Необходимо провести вычислительный эксперимент, позволяющий выявить наиболее оптимальный алгоритм из числа предложенных для решения поставленной задачи.

Для проведения эксперимента предлагается запуск каждого алгоритма на одном и том же массиве вводных данных с последующим замером двух основных параметров: скорость выполнения программы и объём оперативной памяти, занимаемый программой. Каждый алгоритм запускается три раза, после чего вычисляется среднее арифметическое каждого из исследуемых параметров и проводится сравнение эффективности.

Программно-аппаратная конфигурация системы, на которой проводится тестирование, следующая:

– виртуальная машина VMware с операционной системой Linux Ubuntu версии 16.04.7;

– процессор AMD Ryzen 5 3350G с искусственно заниженной тактовой частотой до 1.2 ГГц (для наглядности результатов скорости выполнения программы), виртуальной машине выделен один логический процессор;

– оперативная память DDR4 2400 МГц, выделено для виртуальной машины 1024 Мб.

Для определения скорости выполнения программы используется стандартный встроенный профайлер cProfile. Его использование влияет на скорость выполнения программы, добавляя в среднем +20-30% к времени выполнения [11][16], однако позволяет отследить скорость выполнения каждой функции (рисунок 17).

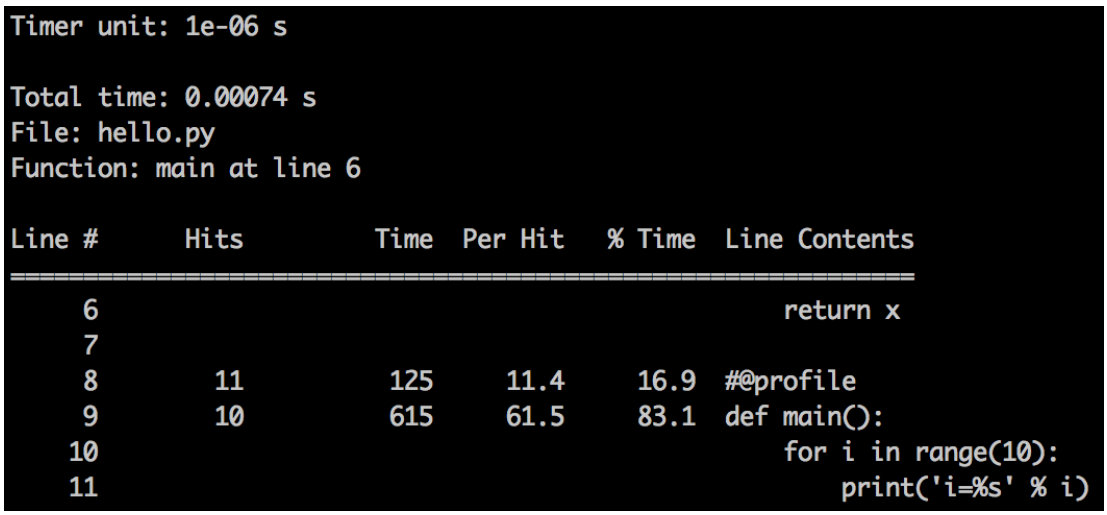

Рисунок 17 – Отчёт cProfile

Важной особенностью cProfile является то, что профайлер отслеживает не только время выполнения, но и число вызовов каждой строки, что помогает в профилировании сложных проектов с многократным вызовом функций, а также в работе с циклами.

Отслеживание оперативной памяти, занимаемой программой, контролирует сторонний профайлер memory_profiler [14]. Он позволяет отслеживать использование памяти как построчно, так и в целом за весь период работы программы, а также показывает, на каком шаге программа требует больше ресурсов, а на каком освобождает память.

Массив вводных данных задаётся один раз для всех алгоритмов при помощи отдельного модуля, листинг которого приведён в Приложении А.

3.2 Проведение эксперимента и анализ результатов

Сформулировав методику проведения испытаний, прейдём к их реализации. На вход каждый алгоритм получает сгенерированный ранее файл с описанием графа и указанными вершинами start и end. Отчёты профайлеров (рисунок 17) фиксируются и на основе полученных из них данных строится сводная таблица.

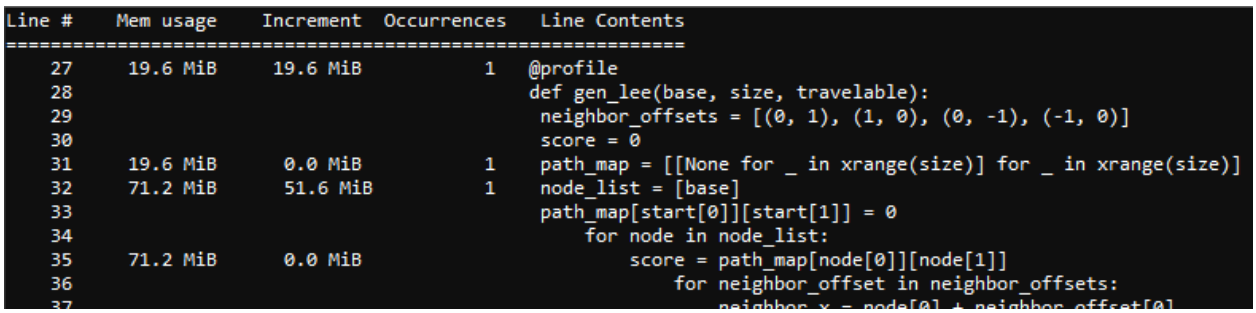

Рисунок 18 – Вывод отчёта memory_profiler на примере алгоритма Ли

После тестирования всех выбранных алгоритмов, мы получили следующие данные (таблица 2).

Таблица 2 – Результаты тестирования алгоритмов

Название	Время выполнения, мс (средние значения)	Объём занимаемой памяти, Мб (средние значения)
Волновой алгоритм	7392	96.3
Жадный алгоритм	6505	81.5
Алгоритм Дейкстры	6744	126.5
Алгоритм A*	5925	140.1

Данные эксперимента подтвердили теоретические предположения об эффективности алгоритмов. Самым быстрым является алгоритм A*, однако он же требует наибольшее количество оперативной памяти для работы. Наиболее медленными ожидаемо являются волновой алгоритм и алгоритм Дейкстры. Жадный алгоритм на практике оказался более оптимальным, чем предполагалось изначально.

Полученные результаты позволяют подтвердить предположение о том, что для решения задачи поиска кратчайшего пути графа наиболее оптимальным является использование алгоритма A*, поскольку он

выполняется быстрее остальных, пускай и ценой большего потребления ресурсов.

Выводы по Главе 3

В данной главе были описаны условия проведения вычислительного эксперимента, его аппаратное и программное обеспечение, исходные данные, охарактеризована методика проведения вычислений.

Затем был проведён вычислительный эксперимент для каждого выбранного ранее алгоритма. Полученные данные о скорости выполнения программы и потреблении программой памяти были сведены в таблицу и проанализированы. Анализ показал, что наиболее оптимальным с точки зрения скорости выполнения является алгоритм A*

Заключение

В ходе работы над бакалаврской работой был проведён анализ предметной области, а также исследований отечественных и зарубежных учёных в целях выяснения уровня проработанности проблемы и её готовых решений. После постановки задачи, из числа алгоритмов поиска кратчайшего пути в графе были выбраны несколько наиболее популярных и зарекомендовавших себя алгоритмов для дальнейшей работы – волновой алгоритм, жадный алгоритм, алгоритм Дейкстры и алгоритм A*.

Для каждого выбранного алгоритма было подготовлено детальное описание принципа работы в целях упрощения перевода в программный код. Алгоритмы были проанализированы на предмет соответствия решаемой задачи, были выявлены их достоинства, недостатки и ограничения, присущие некоторым алгоритмам. Поскольку все перечисленные алгоритмы подходят для решения поставленной задачи, были реализованы программы на языке программирования Python для каждого выбранного алгоритма.

Далее была осуществлена проверка разработанных программ на одинаковых графах с целью выяснения особенностей работы и определения наиболее подходящего для решения поставленной задачи алгоритма. Тестирование осуществлялось в специальной программно-аппаратной среде, для фиксации времени работы программ и объёма используемой ими памяти применялись профайлеры cProfile и memory_profiler. По результатам исследования, наиболее оптимальным для решения проблемы стал алгоритм A* за счёт скорости его работы.

Результаты, полученные в ходе работы над ВКР могут быть использованы в дальнейших исследованиях задачи поиска кратчайшего пути и других похожих задачах теории графов. Программы, разработанные в данной бакалаврской работе, могут быть использованы для практического решения задачи поиска кратчайшего пути в графе.

Список используемой литературы

1. Богомолов А.М. Алгебраические основы теории дискретных систем.— М.: Наука, 1997. — 368 с
2. Гришков Данила Юрьевич, Аусилова Назерке Мырзабековна ЯЗЫК ВЫСОКОГО УРОВНЯ ПРОГРАММИРОВАНИЯ PYTHON // НИР/S&R. 2022. №1 (9). [Электронный ресурс] URL: https://cyberleninka.ru/article/n/yazyk-vysokogo-urovnya-programmirovaniya-python
3. Евстигнеев В. А. Глава 3. Итеративные алгоритмы глобального анализа графов. Пути и покрытия // Применение теории графов в программировании / Под ред. А. П. Ершова. — Москва: Наука. Главная редакция физико-математической литературы, 1985. — С. 138—150. — 352 с.
4. Емеличев, В. А. Лекции по теории графов / В. А. Емеличев [и др]. — М. : Наука, 1990. - 384 с.
5. Изотова Т.Ю. Обзор алгоритмов поиска кратчайшего пути в графе // Новые информационные технологии в автоматизированных системах. 2016. №19. URL: https://cyberleninka.ru/article/n/obzor-algoritmov-poiska-kratchayshego-puti-v-grafe.
6. Изотова Т.Ю. Обзор алгоритмов поиска кратчайшего пути в графе // Новые информационные технологии в автоматизированных системах. 2016. №19.
7. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест — М. : МЦНМО, 2009. - 960с.
8. Оре, О. Теория графов М. : Наука, 1980. - 336 с.
9. Плотников, Подвальный Е. С. Решение задачи поиска оптимального пути между двумя точками на графе с нерегулярным весом ребер // Вестник ВГТУ. 2012. №6. С. 22-26.
10. Свами, М. Графы, сети и алгоритмы / М. Свами, К. Тхуласираман — М. : Мир, 1984. - 455 с.

11. Хайнеман, Д. Алгоритмы. Справочник с примерами на C, C++, Java и Python / Д Хайнеман, Г. Поллис, С. Селков; пер. И. В. Красикова. - 2-е изд. — М. : Диалектика, 2017. - 427 с.

12. Харари, Ф. Теория графов — М. : УРСС, 2003. - 300 с.

13. D. Di Caprio, A. Ebrahimnejad, H. Alrezaamiri, and F. J. Santos-Arteaga, A novel ant colony algorithm for solving shortest path problems with fuzzy arc weights – Alexandria Engineering Journal – Vol. 61, no. 5, P. 3403–3415

14. Dijkstra E. W. A note on two problems in connexion with graphs // Numer. Math / F. Brezzi — Springer Science+Business Media, 1959. — Vol. 1, Iss. 1. — P. 269–271

15. Kőnig, Dénes. Theorie der endlichen und unendlichen Graphen. — Leipzig: Akademische Verlagsgesellschaf, 1936.

16. memory-profiler 0.60.0. Project description. [Электронный ресурс]. URL: https://pypi.org/project/memory-profiler/

17. Rote G. (1990) Path Problems in Graphs. In: Tinhofer G., Mayr E., Noltemeier H., Syslo M.M. (eds) Computational Graph Theory. ComputingSupplementum, Vol. 7, Springer, Vienna.

18. Shabina Banu Mansuri1, Shiv kumar. Comparative Analysis of Path Finding Algorithms, Journal of Computer Engineering (IOSR-JCE), Volume 20, Issue 5, Ver. I (Sep - Oct 2018), PP 38-45.

19. The Global Optimal Algorithm of Reliable Path Finding Problem Based on Backtracking Method / [Liang Shen, Hu Shao, Long Zhang et al.] // Mathematical Problems in Engineering. 2017. Vol. 2017. P. 1–10.

20. The Python Profilers. User Guide. [Электронный ресурс]. URL: https://docs.python.org/3/library/profile.html

Приложение А

Программный код генерации графа для тестирования алгоритмов

```
import copy


def print_map(my_map):
    for row in my_map:
        for cell in row:
            print("{:3}".format(cell), end=' ')
        print()
    print()


def generate_map(width, height):
    my_map = []
    for y in range(height):
        row = []
        for x in range(width):
            row.append(0)
        my_map.append(row)
    return my_map


def add_obstacle(my_map, x0, y0, width, heigth):
    for y in range(y0, y0 + heigth):
        for x in range(x0, x0 + width):
            if position_exists(my_map, x, y):
                my_map[y][x] = -1
```

```
class Location:
    x = 0
    y = 0

    def __hash__(self):
        return hash((self.x, self.y))

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y


def position_exists(my_map, x, y):
    location = Location()
    location.x = x
    location.y = y
    return location_exists(my_map, location)


def location_exists(my_map, location):
    if location.y < 0:
        return False
    if location.x < 0:
        return False
    if location.y >= len(my_map):
        return False
    if location.x >= len(my_map[location.y]):
        return False
    return True
```

```
def set_weight(my_map, location, weight):
    if location_exists(my_map, location):
        if my_map[location.y][location.x] > 0:
            return
        my_map[location.y][location.x] = weight


def get_zero_weight_unique_locations(my_map, locations):
    unique_locations = set()
    for location in locations:
        if my_map[location.y][location.x] == 0:
            unique_locations.add(location)

    return unique_locations


def get_neighbors(my_map, location):
    neighbors = []

    if not location_exists(my_map, location):
        return neighbors

    top = copy.copy(location)
    top.y += 1
    if location_exists(my_map, top):
        neighbors.append(top)

    bottom = copy.copy(location)
```

```
bottom.y -= 1
  if location_exists(my_map, bottom):
     neighbors.append(bottom)

  rigth = copy.copy(location)
  rigth.x += 1
  if location_exists(my_map, rigth):
     neighbors.append(rigth)

  left = copy.copy(location)
  left.x -= 1
  if location_exists(my_map, left):
     neighbors.append(left)

  return neighbors
```