Федеральное государственное бюджетное образовательное учреждение высшего образования «Нижегородский государственный архитектурно-строительный университет» (ННГАСУ)

*Факультет инженерно-экологических систем и сооружений*

*Кафедра информационных систем и технологий*

КУРСОВАЯ РАБОТА

по дисциплине: «Язык программирования Python»

На тему: «Алгоритмы поиска пути и структурное программирование»

Выполнил студент 1 курса гр. ИС-33 Лукьянченко Д.Д.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc136089163)

[Теоретическая часть 4](#_Toc136089164)

[Поиск в ширину (breadth-first search, BFS) 4](#_Toc136089165)

[Поиск в глубину (depth-first search, DFS) 6](#_Toc136089166)

[Алгоритм Дейкстры (Dijkstra) 7](#_Toc136089167)

[Алгоритм А\* (А Star или А «со звездочкой») 8](#_Toc136089168)

[Жадный алгоритм (greedy algorithm) 10](#_Toc136089169)

[Реализация алгоритма 13](#_Toc136089170)

[Пример работы 16](#_Toc136089171)

[Заключение 18](#_Toc136089172)

[Список литературы 19](#_Toc136089173)

[Приложение 1 21](#_Toc136089174)

[Листинг программы 21](#_Toc136089175)

**Введение**

В современном мире, где технологии играют огромную роль в нашей жизни, поиск кратчайшего пути является одной из самых важных задач в программировании. Существует множество алгоритмов поиска пути, каждый из которых имеет свои преимущества и недостатки. В данной курсовой работе мы рассмотрим несколько из них: алгоритм Дейкстры, алгоритм A\*, алгоритм поиска в ширину, алгоритм поиска в глубину и жадный алгоритм.

Мы также рассмотрим принципы структурного программирования, которые позволяют разбивать сложные задачи на более простые и понятные компоненты. В результате, мы получим полное представление о том, как эффективно использовать алгоритмы поиска пути в своих проектах и как создавать структурированный и понятный код.

**Цель работы**: реализовать алгоритмы обхода графа: Дейкстры и А\*, поиск в ширину, поиск в глубину и жадный алгоритм для задачи поиска маршрута в лабиринте.

**Задачи**

* Изучить алгоритмы построения маршрута в графе;
* Выделить особенности реализации, необходимые в конкретной задаче поиска маршрута;
* Подготовить исходные данные: лабиринт, координаты точек для посещения при обходе;
* Реализовать алгоритмы с заданными параметрами;
* Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

**1.Теоретическая часть**

**Алгоритм** — это процедура выполнения определенной задачи. Алгоритм является основополагающей идеей любой компьютерной программы [6].

**Поиск в ширину (breadth-first search, BFS)**

Алгоритм поиска в ширину был разработан независимо Муром и Ли для разных приложений (поиск пути в лабиринте и разводка проводников соответственно) в 1959 и 1961 годах [4].

Алгоритм поиска в ширину заключается в том, чтобы разведать вершины графа послойно, в порядке увеличения расстояния от стартовой ноды.

Рассмотрим на примере и возьмем для этого простой связный ненаправленный граф, ребра которого не имеют весов или временных меток. Слой 0 содержит стартовую вершину. Слой 1 будет содержать множество вершин, которые находятся на расстояние одного ребра от стартовой. Каждый последующий слой будет удаляться от стартовой вершины ровно на одно ребро. Алгоритм разведает сначала все ближайшие к стартовой ноды, затем более удаленные и т.д. и завершит работу, когда будут разведаны все вершины и алгоритм не сможет пройти дальше [3].

Этот алгоритм можно сравнить с поджиганием соседних вершин графа: сначала мы зажигаем одну вершину (ту, из которой начинаем путь), а затем огонь за один элементарный промежуток времени перекидывается на все соседние с ней не горящие вершины. В последствие то же происходит со всеми подожженными вершинами. Таким образом, огонь распространяется «в ширину». В результате его работы будет найден кратчайший путь до нужной клетки [4].

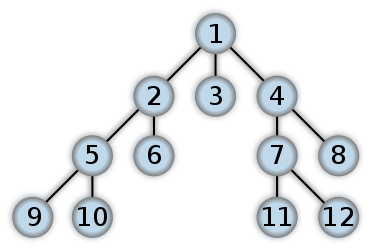
Алгоритм реализуется на основе очереди FIFO (First In, First Out – «Первым пришёл — первым ушёл»), с помощью которого отслеживаются ноды, которые алгоритм уже посещал. Очередь позволяет добавлять объекты в конец списка и удалять объекты из начала за постоянное время.

Рисунок 1 – Поиск в ширину

**Для чего нужен BFS**

1. Для решения задач поиска оптимального пути. Классической задачей считается автоматизированный поиск выхода из лабиринта.
2. Для решения задач, связанных непосредственно с теорией графов, например для поиска компонент связности. Эти задачи в свою очередь решаются в Data Science, теории сетей и электронике.
3. Для задач искусственного интеллекта, связанных с поиском решения с минимальным количеством ходов. В таком случае состояния «умной машины» представляются как вершины, а переходы между ними — как ребра.
4. Для оптимизации памяти при обходе графа в некоторых ситуациях, например для некоторых специфических структур.
5. Для работы с информацией в определенных структурах данных, таких как деревья. Их тоже можно обходить с помощью алгоритма BFS, потому что они — подвид графов [10].

**Поиск в глубину (depth-first search, DFS)**

Алгоритм поиска в глубину отличается от поиска в ширину более агрессивным продвижением по графу [3]. DFS начинает работу в заданной точке, на каждом шаге проходит по лабиринту до следующего поворота и выбирает направление. Если путь оказывается тупиковым, алгоритм возвращается к предыдущему повороту и пробует новое направление. В результате рано или поздно находится нужный путь [11].

Как и BFS, DFS помечает ноду каждый раз, как ее обнаруживает. На каждой итерации алгоритм обходит в произвольном порядке ноды, ближайшие к текущей. На первой же найденной вершине алгоритм будет пытаться найти ближайшие ноды к уже разведанной (в этом он отличается от BFS, который исследует ноды, ближайшие к стартовой) и будет делать это на каждой последующей итерации до тех пор, пока не окажется в ноде, из которой ему некуда уйти. Тогда алгоритм отступает назад и пытается продвинуться дальше по другому пути. Алгоритм так же останавливается, когда все доступные ноды будут разведаны.

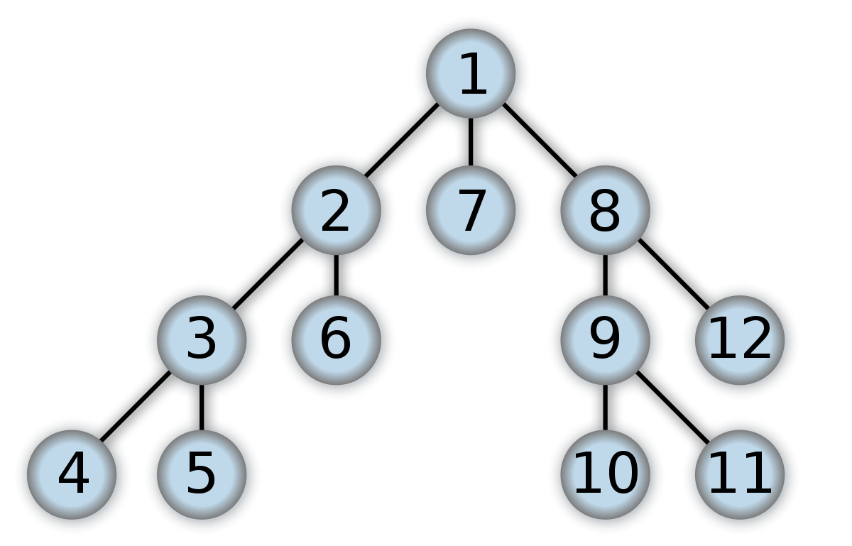
DFS реализуется на основе стека LIFO (last in, first out, «последним пришёл — первым ушёл») [3].

Рисунок 2 – Поиск в глубину

**Для чего нужен алгоритм DFS**

1. Для поиска любого маршрута в лабиринте. В отличие от алгоритма BFS, поиск в глубину ищет не самый короткий, а случайный путь. Правило прохождения лабиринта в реальной жизни “Идти с левой рукой на стене и всегда поворачивать влево” — пример DFS вне программирования.
2. Для решения задач, связанных с построением маршрута: в сети, на карте, в сервисах покупки билетов и так далее. При этом непосредственно для поиска DFS используется не так часто — он чаще нужен для исследования топологии графа.
3. Как составная часть расчетов в более сложных алгоритмах, например для определения максимального транспортного потока.
4. Для решения ряда задач из теории графов, которые используются в программировании и математике: поиска циклов, сортировки и так далее. Мы подробно поговорим об этом ниже [11].

**Алгоритм Дейкстры (Dijkstra)**

Этот алгоритм назван по имени создателя и был разработан в 1959 году. В процессе выполнения алгоритм проверит каждую из вершин графа, и найдет кратчайший путь до исходной вершины. Стандартная реализация работает на взвешенном графе — графе, у которого каждый путь имеет вес, т.е. «стоимость», которую надо будет «заплатить», чтобы перейти по этому ребру. При этом в стандартной реализации веса неотрицательны. На клетчатом поле вес каждого ребра графа принимается одинаковым (например, единицей) [4].

Этот алгоритм имеет два преимущества по сравнению с поиском в ширину: он принимает во внимание стоимость или длину пути и обновляет узлы, если к ним найден лучший путь. Для реализации, список Open с очередью FIFO заменяется приоритетной очередью, где извлеченный узел имеет лучшее значение - здесь, это наименьшая стоимость пути от старта. Однако, он имеет слабость поиска в ширину, игнорируя направление к цели [5].

Алгоритм Дейкстры пошаговый. Сначала выбирается точка, от которой будут отсчитываться пути. Затем алгоритм поочередно ищет самые короткие маршруты из исходной точки в другие. Вершины, где он уже побывал, отмечает посещенными. Алгоритм использует посещенные вершины, когда рассчитывает пути для не посещённых.

**Для чего нужен алгоритм Дейкстры**

Основная задача — поиск кратчайшего пути по схеме, где множество точек соединено между собой отрезками. В виде такой схемы можно представить многие объекты реального мира, поэтому практических примеров использования алгоритма много:

1. автоматическое построение маршрута на онлайн-карте;
2. поиск системой бронирования наиболее быстрых или дешевых билетов, в том числе с возможными пересадками;
3. моделирование движения робота, который перемещается по местности;
4. разработка поведения неигровых персонажей, создание игрового ИИ в геймдеве;
5. автоматическая обработка транспортных потоков;
6. маршрутизация движения данных в компьютерной сети;
7. расчет движения тока по электрическим цепям [1].

**Алгоритм А\* (А Star или А «со звездочкой»)**

Впервые описан в 1968 году Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем. Данный алгоритм является расширением алгоритма Дейкстры, ускорение работы достигается за счет эвристики — при рассмотрении каждой отдельной вершины переход делается в ту соседнюю вершину, предположительный путь из которой до искомой вершины самый короткий. При этом существует множество различных методов подсчета длины предполагаемого пути из вершины. Результатом работы также будет кратчайший путь [4].

Алгоритм А\* обладает двумя ключевыми характеристиками алгоритмов такого рода: оптимальность и полнота. Если алгоритм поиска характеризуется как оптимальный, значит он гарантирует получение лучшего из возможных решений. А когда среди характеристик присутствует определение «полный», это означает, что алгоритм всегда находит решение, если таковое существует [2].

**Как работает алгоритм «A Star»**

Посетив одну конкретную вершину, алгоритм «A Star» перед переходом к следующей исследует все соседние вершины. Все вершины алгоритм разделяет на 3 категории:

1. Неизвестные вершины. Это те, которые не были еще посещены и пока что даже не найдены. Получается, что и путь к ним пока остается загадкой. Таким образом, изначально все вершины, кроме стартовой, будут в этой категории.
2. Известные вершины. Это те вершины, о которых уже известно алгоритму и уже даже известен путь к ним. Такие вершины сохраняются в «списке алгоритма» и становятся в очередь для их посещения и исследования. Из этого списка исследуются те вершины, которые считаются наиболее перспективными.
3. Исследованные вершины. В эту категорию попадают те вершины, которые уже были посещены алгоритмом «A Star». К этим вершинам известен самый короткий путь, поэтому они попадают в «закрытый список» — этот список нужен для того, чтобы исключить многократное исследование одних и тех же вершин.

Когда одна из вершин становится полностью исследованной, она попадает в категорию «исследованные вершины», а все ее соседи попадают в категорию «известные вершины» и становятся годными для исследования. На каждой уже исследованной вершине устанавливается указатель до той уже исследованной вершины, к которой у нее будет кратчайший путь.

Алгоритм «A Star» завершает свою работу только в том случае, если конечная вершина переносится в категорию «исследованные вершины». В этом случае уже будет весь список исследованных вершин, а на каждой из них будет стоять указатель с кратчайшим путем. Поэтому несложно будет по указателям отследить кратчайший путь от конечной вершины до начальной.

Алгоритм «A Star» находит кратчайший путь между вершинами, основываясь на стоимости и «весе» ребер. Поэтому путь, который находит «A Star», можно по праву назвать «самым быстрым» или «самым простым». По этой причине алгоритм «A Star» очень часто применяется как раз для планирования кратчайших путей, его также часто применяют в играх.

Однако его применение ограничивается его главным недостатком — потребностью в большом количестве памяти, потому что алгоритм «A Star» хранит всю информацию об известных и исследованных вершинах [9].

**Жадный алгоритм (greedy algorithm)**

Жадный алгоритм (greedy algorithm) — это алгоритм, который на каждом шагу делает локально наилучший выбор в надежде, что итоговое решение будет оптимальным. К примеру, алгоритм Дейкстры нахождения кратчайшего пути в графе вполне себе жадный, потому что мы на каждом шагу ищем вершину с наименьшим весом, в которой мы еще не бывали, после чего обновляем значения других вершин. При этом можно доказать, что кратчайшие пути, найденные в вершинах, являются оптимальными [7].

Чтобы решить проблему, основанную на жадном подходе, есть два этапа:

1. сканирование списка предметов
2. оптимизация.

Эти этапы выполняются параллельно по ходу деления массива. Есть два условия, которые определяют жадную парадигму:

* Каждое поэтапное решение должно структурировать проблему в направлении ее наиболее приемлемого решения.
* Достаточно если структурирование проблемы может остановиться за конечное число жадных шагов.

Логика работы жадного алгоритма в ее самой простой форме сводилась к «жадному» или «не жадному». Эти утверждения были определены подходом, принятым для продвижения на каждой стадии алгоритма.

Алгоритм перестает быть жадным, если на любом этапе он делает шаг, который не является локально жадным. Проблема останавливается без дальнейшего охвата жадности [8].

**Для чего нужен жадный алгоритм**

Вот причины использования жадного подхода:

1. У жадного подхода есть несколько компромиссов, которые могут сделать его пригодным для оптимизации.
2. Одна из важных причин состоит в том, чтобы немедленно найти наиболее выполнимое решение. В задаче выбора действий (поясняется ниже), если до завершения текущего действия можно выполнить больше действий, эти действия могут быть выполнены в одно и то же время.
3. Другая причина заключается в рекурсивном разделении проблемы на основе условия без необходимости объединения всех решений.
4. В задаче выбора действий шаг «рекурсивное разделение» достигается путем сканирования списка элементов только один раз и рассмотрения определенных действий.

**Как решить проблему выбора активности**

В примере планирования действий есть время «начала» и «окончания» для каждого действия. Каждое мероприятие индексируется номером для справки. Есть две категории деятельности.

* **рассматриваемая деятельность:** это деятельность, которая является ссылкой, с помощью которой анализируется способность выполнять более одной оставшейся деятельности.
* **оставшиеся действия:** действия по одному или нескольким показателям перед рассматриваемой деятельностью.

Общая продолжительность дает стоимость выполнения действия. То есть (финиш – старт) дает нам длительность как стоимость деятельности. Вы узнаете, что жадная степень – это количество оставшихся действий, которые вы можете выполнить во время рассматриваемой деятельности [8].

**2. Реализация алгоритма**

Импортирую класс PriorityQueue из модуля queue. PriorityQueue — это очередь с приоритетами, которая хранит элементы в порядке возрастания их приоритета.

Функция read\_maze читает файл с именем filename и возвращает двумерный список (матрицу), представляющую лабиринт. Функция использует конструкцию генератора списка для создания списка символов в каждой строке файла.

Функция get\_neighbors принимает матрицу лабиринта и координаты клетки в виде кортежа (row, col) и возвращает список соседних клеток, которые можно достичь из данной клетки. Функция проверяет, что соседние клетки находятся внутри границ лабиринта и не являются стенами (обозначенными символом «#»).

Функция bfs использует алгоритм поиска в ширину (BFS) для поиска кратчайшего пути в лабиринте, заданном в виде двумерного списка. Функция начинает поиск с клетки (0, 1), которая является стартовой точкой, и заканчивает поиск на клетке (len(maze) - 1, len(maze[0]) - 2), которая является точкой выхода из лабиринта. Алгоритм BFS использует очередь для хранения текущих вершин и списка посещенных вершин. На каждой итерации алгоритма извлекается первый элемент из очереди и проверяется, является ли он конечной точкой. Если да, то возвращается путь до этой точки. Если текущая вершина не является конечной, то она добавляется в список посещенных вершин, и для каждого соседа текущей вершины, который еще не был посещен, добавляется новый путь в очередь. А если путь до конечной точки не найден, то функция возвращает None.

get\_heuristic — это функция, которая принимает координаты двух клеток в виде кортежей (cell и end) и возвращает евклидово расстояние между ними. Она используется в алгоритме A\* для оценки расстояния от текущей клетки до конечной клетки.

Далее я реализую алгоритм A\* для поиска кратчайшего пути в лабиринте. Он начинается с определения начальной и конечной клеток, создания очереди с приоритетом, которая будет использоваться для хранения текущих позиций и путей, а также множества посещенных клеток. Затем алгоритм выполняется в цикле, пока очередь не опустеет. На каждой итерации извлекается элемент из очереди с наименьшим приоритетом (это определяется суммой длины пути до текущей клетки и эвристической оценки расстояния до конечной клетки). Если текущая клетка совпадает с конечной, то алгоритм завершается и возвращается найденный путь. Иначе, текущая клетка добавляется в множество посещенных, и для каждого соседа текущей клетки, который еще не был посещен, создается новый путь, который включает этого соседа. Этот новый путь добавляется в очередь с приоритетом, и процесс продолжается до тех пор, пока не будет найден путь или очередь не опустеет. Если путь не найден, возвращается значение None.

Функция solve\_maze использует алгоритм поиска в ширину (BFS) для поиска кратчайшего пути в лабиринте, заданном в файле "maze-for-u.txt". Функция сначала считывает лабиринт из файла с помощью функции read\_maze, затем вызывает функцию bfs для поиска кратчайшего пути. После этого функция изменяет лабиринт, заменяя все клетки на пути к выходу на символ «.». Затем результат записывается в файл «maze-for-me-done1.txt». Потом используется алгоритм A\* для поиска кратчайшего пути в лабиринте, заданном в файле «maze-for-u.txt». Функция сначала считывает лабиринт из файла с помощью функции read\_maze, затем вызывает функцию a\_star для поиска кратчайшего пути. После этого функция изменяет лабиринт, заменяя все клетки на пути к выходу на символ «,» и результат записывается в файл «maze-for-me-done2.txt».

В итоге функция solve\_maze() решает задачу поиска выхода из лабиринта и записывает результат в файлы.

**Пример работы**

На вход подаётся файл «maze-for-u.txt», который был сгенерирован в результате работы «gen\_lab\_origin.py». «maze-for-u.txt» представляет собой текстовый файл с лабиринтом, состоящий из пробелов и «#». Где пробелы являются свободными клетками, по которым может перемещаться аватар, а «#» - являются стенами, сквозь которые проходить нельзя.

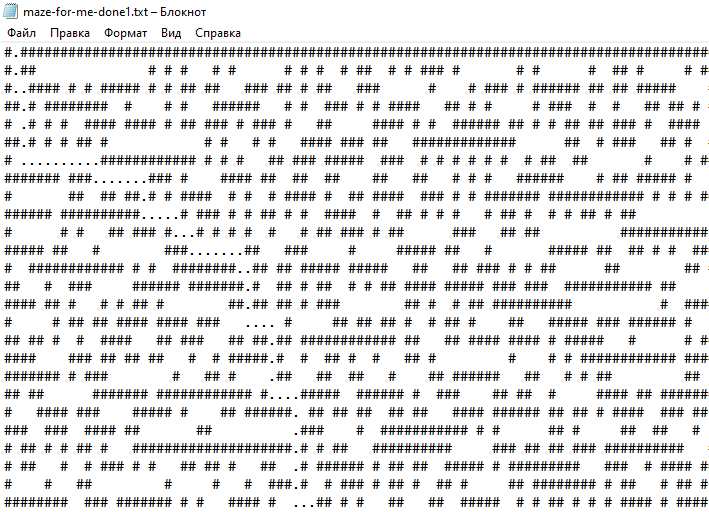
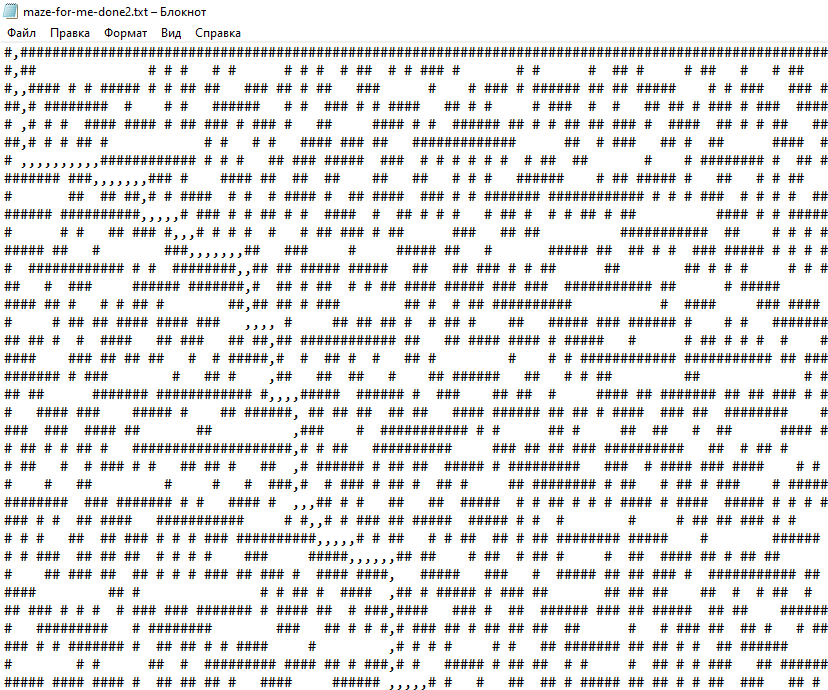
 В результате работы моего кода (приложение 1), будет создано 2 новых файла: «maze-for-me-done1.txt» и «maze-for-me-done2.txt» на основе файла «maze-for-u.txt». В файле «maze-for-me-done1.txt» построен путь он входа в лабиринт до его выхода при помощи точек «.», которые заменят пробелы, используя алгоритм поиска пути в ширину. В файле «maze-for-me-done2.txt» построен путь от входа в лабиринт до его выхода при помощи запятых «,», которые заменят пробелы, используя алгоритм поиска пути А\* (A Star). Точки «.» и запятые «,» — это те клетки, которые образуют путь, по которому прошёл наш аватар от входа в лабиринт до его выхода.

Рисунок 3 – Пример работы моей программы. Часть файла «maze-for-me-done1.txt»

Рисунок 4 – Пример работы моей программы. Часть файла «maze-for-me-done2.txt»

**Заключение**

В результате выполнения курсовой работы были успешно реализованы и протестированы два алгоритма поиска оптимального пути в лабиринте — поиск в ширину и алгоритм A\* для поиска выхода из лабиринта любого размера при условии, что у этого лабиринта есть вход и выход.

Также были изучены принципы работы каждого из алгоритмов, а также способы их применения для решения задачи поиска пути. Были написаны функции для определения местоположения элементов лабиринта, и сами алгоритмы поиска пути, а также результаты работы программы были сохранены в файлы «maze-for-me-done1.txt» и «maze-for-me-done2.txt».»

В моём случае для большого лабиринта с размером 600 строк на 800 символов в строке, алгоритм A\* оказался более эффективным, так как он использует эвристику для принятия решений и может находить оптимальный путь быстрее, чем поиск в ширину. Однако, если лабиринт имеет простую структуру и небольшой размер, то поиск в ширину может быть быстрее и проще в реализации.

Таким образом, выполнение курсовой работы позволило не только углубить знания в области алгоритмов поиска пути и структурного программирования, но и практически применить их при помощи языка программирования Python. Результаты работы могут быть использованы для решения подобных задач в будущем.

**Список литературы**

1. Алгоритм Дейкстры // blog.skillfactory.ru URL: <https://blog.skillfactory.ru/glossary/algoritm-dejkstry/> (дата обращения: 25.04.2023).
2. Алгоритм поиска A\* // medium.com URL: <https://medium.com/nuances-of-programming/алгоритм-поиска-a-3bb59be05a79> (дата обращения: 25.04.2023).
3. Алгоритмы поиска в ширину и в глубину // konstantinklepikov.github.io URL: <https://konstantinklepikov.github.io/2020/10/09/breadth-first-and-depth-first-search-algorithms.html> (дата обращения: 25.04.2023).
4. Алгоритмы поиска пути в графе // tproger.ru URL: <https://tproger.ru/articles/pathfindings/> (дата обращения: 25.04.2023).
5. Алгоритмы поиска пути. // pmg.org.ru URL: <http://pmg.org.ru/ai/stout.htm#detour> (дата обращения: 25.04.2023).
6. Алгоритмы. Руководство по разработке. 3-е изд. // anyflip.com URL: <https://coderlessons.com/tutorials/kompiuternoe-programmirovanie/osnovy-algoritmov/1-zhadnyi-algoritm> (дата обращения: 25.04.2023).
7. Жадные алгоритмы // habr.com URL: <https://habr.com/ru/articles/120343/> (дата обращения: 25.04.2023).
8. Жадный алгоритм // coderlessons.com URL: <https://coderlessons.com/tutorials/kompiuternoe-programmirovanie/osnovy-algoritmov/1-zhadnyi-algoritm> (дата обращения: 25.04.2023).
9. Поисковый алгоритм A Star: что это и как эффективно его использовать? // codernet.ru URL: <https://codernet.ru/articles/web/poiskovyij_algoritm_a_star_chto_eto_i_kak_effektivno_ego_ispolzovat/> (дата обращения: 25.04.2023).
10. BFS // blog.skillfactory.ru URL: <https://blog.skillfactory.ru/glossary/bfs/> (дата обращения: 25.04.2023).
11. DFS // blog.skillfactory.ru URL: <https://blog.skillfactory.ru/glossary/dfs/> (дата обращения: 25.04.2023).

# Приложение 1

## Листинг программы

from math import sqrt  
from queue import PriorityQueue  
  
  
# Функция, которая считывает лабиринт из текстового файла и возвращает его в виде двумерного массива  
def read\_maze(filename):  
 with open(filename) as file:  
 maze = [[char for char in line.strip()] for line in file]  
 return maze  
  
  
# Функция, которая возвращает список соседних клеток, в которые можно перейти аватару  
def get\_neighbors(maze, cell: tuple[int, int]):  
 row, col = cell  
 neighbors = [(row - 1, col), (row + 1, col), (row, col - 1), (row, col + 1)]  
 valid\_neighbors = []  
 for neighbor in neighbors:  
 row, col = neighbor  
 if 0 <= row < len(maze) and 0 <= col < len(maze[0]) and maze[row][col] != "#":  
 valid\_neighbors.append(neighbor)  
 return valid\_neighbors  
  
  
# Поиск пути от начальной точки до конечной точки в лабиринте через поиск в ширину  
def bfs(maze):  
 start = (0, 1)  
 end = (len(maze) - 1, len(maze[0]) - 2)  
 queue = [(start, [start])]  
 visited = set()  
 while queue:  
 current, path = queue.pop(0)  
 if current == end:  
 return path  
 visited.add(current)  
 for neighbor in get\_neighbors(maze, current):  
 if neighbor not in visited:  
 queue.append((neighbor, path + [neighbor]))  
 return None  
  
  
# Реализация алгоритма А\*  
# Вычисляет эвристическое расстояние от ячейки до конечной точки  
def get\_heuristic(cell, end):  
 return sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2)  
  
  
# Функция, которая ищет путь от начальной точки до конечной точки в лабиринте, используя алгоритм A\*  
def a\_star(maze):  
 start = (0, 1)  
 end = (len(maze) - 1, len(maze[0]) - 2)  
 queue = PriorityQueue()  
 queue.put((0, start, [start]))  
 visited = set()  
 while not queue.empty():  
 p, current, path = queue.get()  
 if current == end:  
 return p, path  
 visited.add(current)  
 for neighbor in get\_neighbors(maze, current):  
 if neighbor not in visited:  
 new\_path = path + [neighbor]  
 priority = len(new\_path) + get\_heuristic(neighbor, end)  
 queue.put((priority, neighbor, new\_path))  
 return None  
  
  
# Вывод результатов  
def solve\_maze():  
 filename = "maze-for-u.txt"  
 maze = read\_maze(filename)  
  
 # Cоздание выходного файла для поиска выхода из лабиринта через алгоритм поиска в ширину  
 path1 = bfs(maze)  
 for place in path1:  
 maze[place[0]][place[1]] = "."  
 result\_1 = ""  
 for line in maze:  
 result\_1 += "".join(line) + "\n"  
 with open("maze-for-me-done1.txt", "w") as f:  
 f.write(result\_1)  
  
 # Создание выходного файла для поиска выхода из лабиринта через алгоритм А\*  
 path\_2 = a\_star(maze)  
 path\_2 = path\_2[1]  
  
 for place in path\_2:  
 maze[place[0]][place[1]] = ","  
  
 result\_2 = ""  
 for line in maze:  
 result\_2 += "".join(line) + "\n"  
  
 with open("maze-for-me-done2.txt", "w") as f:  
 f.write(result\_2)  
  
  
solve\_maze()