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 Назарова Ю. В.

Проверил Морозов Н. С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc1)

[Задачи 3](#_Toc2)

[1.Теоретическая часть 4](#_Toc3)

[2. Реализация алгоритма 7](#_Toc4)

[Пример работы 14](#_Toc5)

[Заключение 14](#_Toc6)

[Список литературы 15](#_Toc7)

[Приложение 1 16](#_Toc8)

[Листинг программы 16](#_Toc9)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании. Графы превратились в невероятно сильное средство моделирования и получения данных из соцсетей, веб-страниц и ссылок, а также определения местоположения и маршрутов в GPS. Любой набор объектов, которые связаны друг с другом, можно сейчас представить с помощью графа. Обход или поиск — это одна из фундаментальных операций, выполняемых на графах. [1] Одним из таких является поиск в глубину. Поиск в глубину начинается с определённой вершины, затем уходит как можно дальше вдоль каждой ветви и возвращается обратно. Здесь тоже необходимо отслеживать посещённые алгоритмом вершины. Для того, чтобы стало возможным возвращение обратно, при реализации алгоритма поиска в глубину используется структура данных «стек». Поиск в глубину применяется, например, в головоломках с единственным решением (например, лабиринтах). [1]

**Цель работы**: реализовать алгоритмы обхода графа: в глубину(dfs) и А\* 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 DFS (поиск в глубину).**

Поиск в глубину начинается с определённой вершины, затем уходит как можно дальше вдоль каждой ветви и возвращается обратно. Здесь тоже необходимо отслеживать посещённые алгоритмом вершины. Для того, чтобы стало возможным возвращение обратно, при реализации алгоритма поиска в глубину используется структура данных «стек».

Классический пример использования алгоритма — поиск случайного пути в лабиринте. 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.

Применяется:

* для нахождения пути между двумя вершинами;
* для обнаружения циклов на графе;
* в топологической сортировке;
* в головоломках с единственным решением (например, лабиринтах).

Задача поиска выхода из лабиринта известна с древних времен. В терминах графов ее можно формализовать так: лабиринт - это неориентированный граф, вершины которого представляют "перекрестки" лабиринта, а ребра - дорожки между соседними перекрестками. Одна или несколько вершин отмечены как выходы. Задача состоит в построении пути из некоторой исходной вершины в вершину-выход.

Мы рассмотрим метод обхода всех вершин графа, называемый поиском в глубину. Его идею кратко можно описать так:

Находясь в некоторой вершине v, идем из нее в произвольную еще не посещенную смежную вершину w, если такой вершины нет, то возвращаемся в вершину, из которой мы пришли в v.

Сколько ресурсов требуется алгоритму?

Алгоритм просматривает каждое ребро один раз, и выполняет для каждой вершины константное число действий. Обозначая число вершин как W, а ребер — как V, получаем, что время работы — O(W+V).

Нерекурсивные реализации.

Так как рекурсия отнимает много программных ресурсов, есть несколько реализаций без ее использования. Они сложнее в написании, но могут сэкономить время работы программы.

Самый простой вариант — по-особому хранить в памяти посещенные вершины. Для этого используется такая структура данных, как стек, — о ней можно прочитать в нашей статье. В стек поочередно помещаются непосещенные вершины-соседки, и тот используется как “карта” для будущих посещений. Этот способ тоже нагружает программные ресурсы, но не так сильно.

Второй вариант — хранить в стеке не сами вершины, а их номера и номера смежных с ними вершин. Это сложнее и, по сути, является имитацией так называемого стека вызовов — участка памяти, где хранятся вызванные функции. Именно его может перегрузить рекурсия.

Третий способ — реализовать сам граф на указателях: помещать в каждой вершине указатели на предыдущие и номера смежных вершин.

**Алгоритм A\*. Поиск кратчайшего пути.**

Поиск короткого пути — это то, чем каждый занимается ежедневно. Алгоритм А \*— один из самых популярных методов решения задач на поиск кратчайшего маршрута. Его относят к информированным алгоритмам поиска, так как для решения задач используются данные о стоимости пути и принципы эвристики.

A \* (произносится как "A-star") - это алгоритм обхода графа и по пути, который используется во многих областях информатики благодаря своей полноте, оптимальности и оптимальной эффективности. Одним из основных практических недостатков является его O(b^d) пространственная сложность, поскольку он хранит все сгенерированные узлы в памяти. Таким образом, в практических системах маршрутизации путешествий его обычно превосходят алгоритмы, которые могут предварительно обрабатывать график для достижения лучшей производительности, а также подходы, ограниченные объемом памяти; однако A \* по-прежнему является лучшим решением во многих случаях.

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.

Для понимания работы алгоритма А\* необходимо владеть следующими терминами:

* Узел (или вершина) — все потенциальные уникальные позиции или остановки.
* Переход — само перемещение между вершинами или узлами.
* Начальный узел — тот, от которого начинается путь.
* Конечный узел — тот, в котором путь должен завершиться.
* Пространство поиска — коллекция всех допустимых узлов.
* Стоимость — числовое значение (например, расстояние, время или денежная стоимость), характеризующее отрезок пути от одного узла к другому.
* g(n) — стоимость пути от начальной вершины до любой другой.
* h(n) — эвристическое приближение стоимости пути от узла n до конечного узла.
* f(n) — минимальная стоимость перехода в соседний узел.\

A \* - это алгоритм обоснованного поиска, или поиск по принципу наилучшего начала, что означает, что он сформулирован в терминах взвешенных графов: начиная с определенного начального узла графика, он направлен на поиск пути к заданному целевому узлу с наименьшими затратами (наименьшее пройденное расстояние, наименьшее время и т.д.). Он делает это, поддерживая дерево путей, берущих начало в начальном узле, и расширяя эти пути по одному краю за раз, пока не будет удовлетворен критерий завершения.

На каждой итерации своего основного цикла A \* необходимо определить, какой из его путей следует расширить. Он делает это на основе стоимости пути и оценки затрат, необходимых для расширения пути вплоть до цели.

Каждый раз при посещении узла подсчитывается его стоимость f(n) (за n принимается соседний узел). Таким образом, алгоритм посещает все соседние узлы и высчитывает тот, у которого данный показатель минимален.

Формула выглядит следующим образом:

f(n) = g(n) + h(n)

**Заключение**

Алгоритм A \* — это мощный алгоритм в сфере ИИ с широким спектром применения. Это самый популярный способ для нахождения кратчайшего пути, так как система реализации чрезвычайно гибкая. Сегодня этот алгоритм применяют в различных областях от машинного обучения до разработки игр, например, при выстраивании маршрута по сложной местности с препятствиями.

# 2. Реализация алгоритма

**1. Вспомогательные функции**, которые помогут для дальнейшей реализации алгоритмов поиска и построения маршрута.

На начальном этапе мне понадобится несколько простых функций:

* Функция create\_temple (рис. 1.1) создает двумерный массив нужного нам размера, заполненный нулями. Это понадобится нам для создания шаблонов под графы.

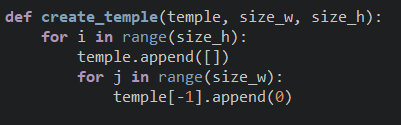


рис. 1.1. Функция create\_temple

* Функция get\_coord (рис. 1.2) считывает от пользователя координаты (Аватара, выхода, ключа) попутно проверяя корректность координат. Проверка на корректность заключается в том, чтобы введенные пользователем координаты не вышли за пределы лабиринта, и не были координатами где находится стена.

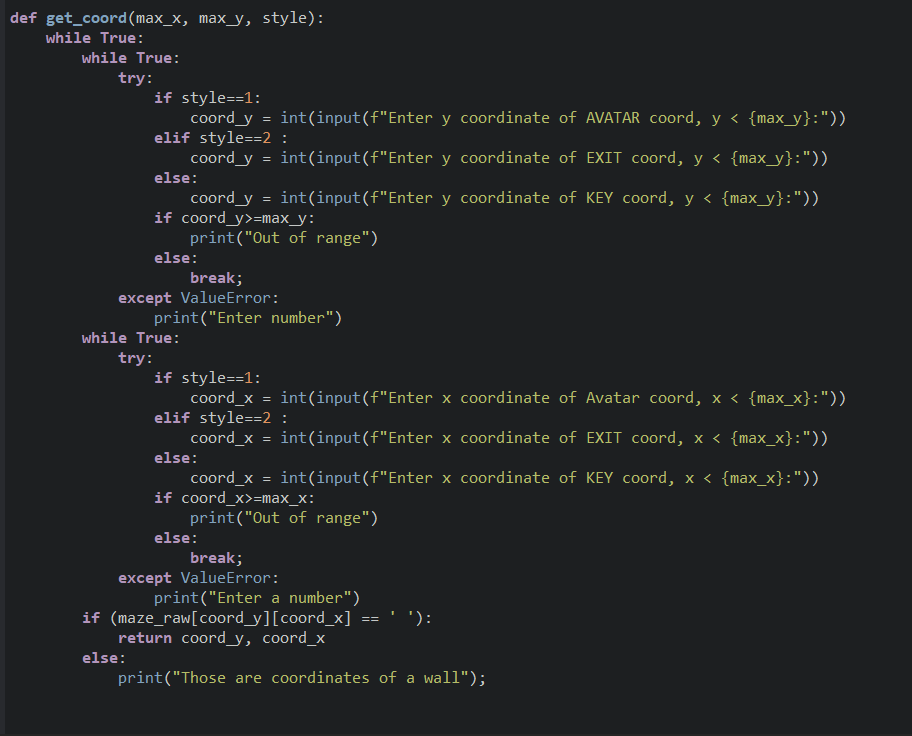


рис. 1.2. Функция get\_coord

* Функция crawl\_graph (рис. 1.3) возвращает уже пройденный(решенный) нами граф в изначальный вид. На вход поступает граф представляющий собой двумерный массив из 0, 1 и 3. Где 1-стена, 0-проход, 3-путь аватара. То есть crawl\_graph из числового представления лабиринта делает текстовое(изначальное).

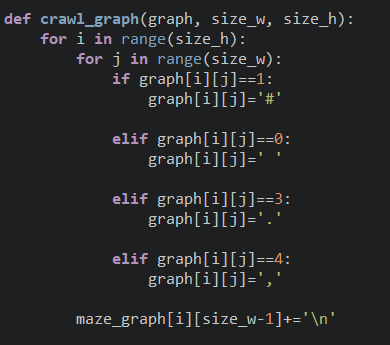


рис. 1.3. Функция crawl\_graph

**2.Реализация алгоритма поиска в глубину (Depth-First-Search, DFS).**

Идея Алгоритма поиска пути DFS:

1. Сделайте начальную клетку текущей и отметьте ее как посещенную.

2. Пока не найден выход

  1. Если текущая клетка имеет не посещённых «соседей»

    1. Протолкните текущую клетку в стек

    2. Выберите случайную клетку из соседних

    3. Сделайте выбранную клетку текущей и отметьте ее как посещенную.

  2. Иначе если стек не пуст

    1. Выдерните клетку из стека

    2. Сделайте ее текущей

  3. Иначе выхода нет

Критерий нахождения «выхода» очень прост: достаточно сравнить координаты текущей точки и координаты «выхода»: если они равны, путь между стартовой и выходной точками найден.

Функция вернет значение True, если путь был найден, а также изменит глобальную переменную stack, в которой будет хранится координаты пути. Или функция вернет False, если путь не был найден. (рис. 2.1)

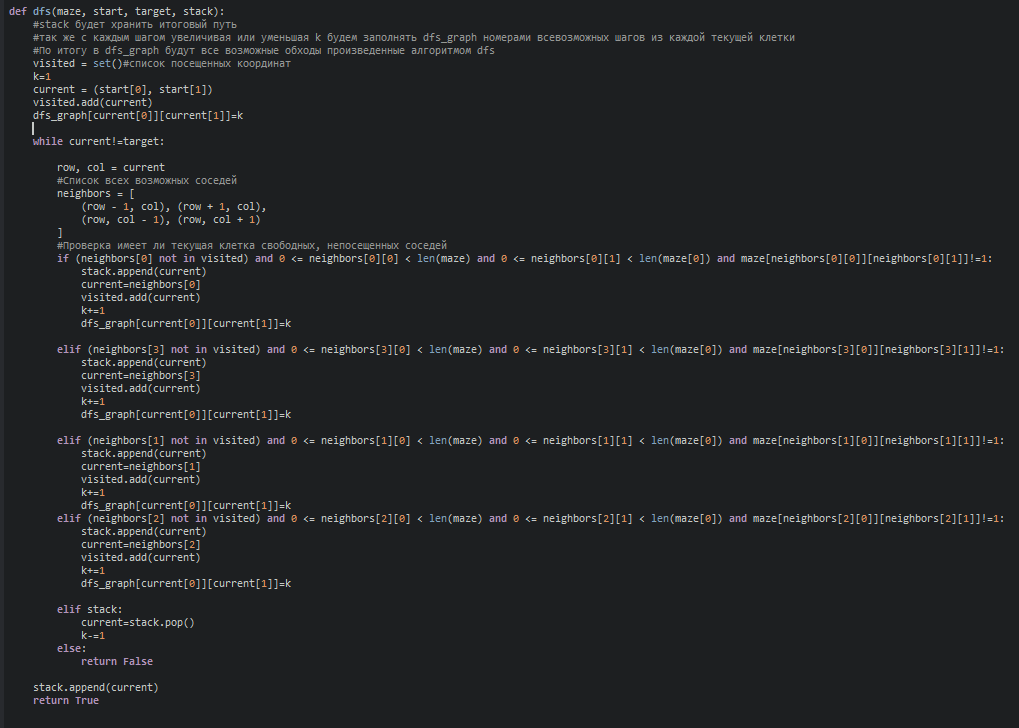


рис. 2.1. Реализация алгоритма поиска в глубину (Depth-First-Search, DFS).

Итак, когда созданы все необходимые функции приступим к реализации тела программы(main).

**1 Этап**. Считываем лабиринт с файла и находим его размеры. (рис. 2.2)

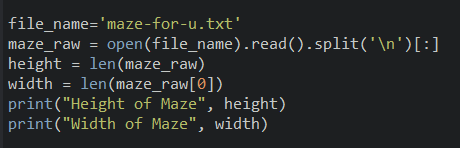


рис. 2.2. Считываем лабиринт с файла и находим его размеры.

Просим пользователя ввести координаты и обрабатываем их. (рис. 2.3)

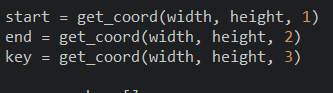


рис. 2.3. Ввод и обработка координат.

Создаем граф и преобразуем его текстовое представление в виде двумерного массива 0 и 1. (рис. 2.4)

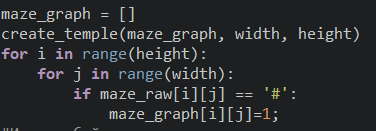


рис. 2.4. Создание и преобразование графа.

**2 Этап**. Ищем любой путь от старта до ключа с помощью алгоритма dfs(поиск в глубину). И изменяем исходный граф добавив в него путь от старта до ключа. (рис. 2.5)

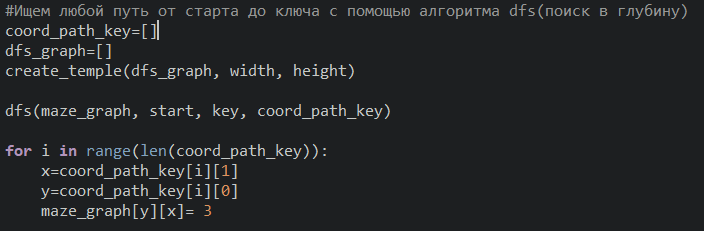


рис. 2.5. Поиск оптимального пути от входа до ключа с помощью поиска в ширину.

**3 Этап**. Ищем оптимальный путь от ключа до выхода с помощью алгоритма A\*. И изменяем исходный граф добавив в него путь от ключа до Выхода. (рис. 2.6)

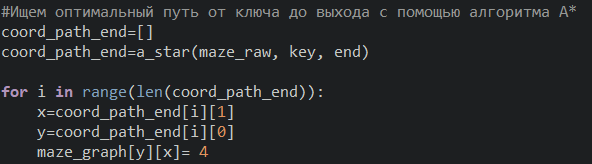


рис. 2.6. Поиск оптимального пути от входа до ключа с помощью А\*.

**4 Этап**. Граф готов. Меняем его представление для печати его в txt файл. (рис. 2.7)

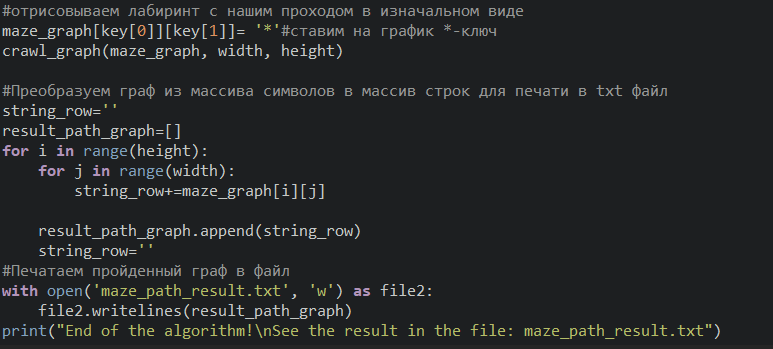


рис. 2.7. Изменение готового графа для печати в txt файл.

**3. Реализация алгоритма A\*.**

Вспомогательная функция available\_paths (рис. 3.1) смотрит все возможные пути из текущей точки (всех свободных соседей) и добавляет их в координаты в список available(доступные).

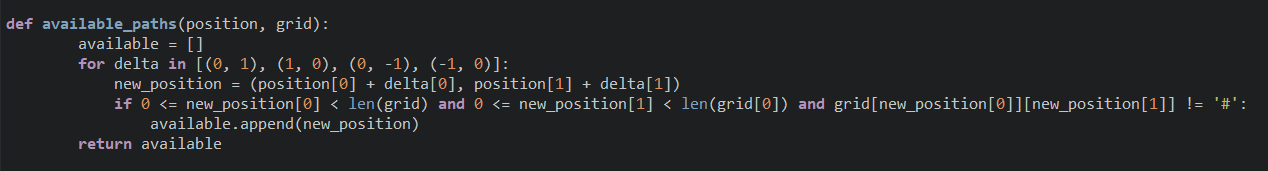
****

рис. 3.1. Функция available\_paths.

Также понадобится эвристическая функция (рис. 3.2), сообщающая нам, насколько мы близки к цели.

****

рис. 3.2. Эвристическая функция heuristic.

И наконец, сам алгоритм a\_star (рис. 3.3), который возвращает оптимальный путь, или возвращает None если путь не найден. Для реализации данного алгоритма понадобилась библиотека heapq. Модуль heapq обеспечивает реализацию алгоритма очереди кучи, также известного как алгоритм очереди приоритетов. Функция heappop() модуля `heapq` возвращает и удаляет наименьший элемент из кучи , сохраняя инвариант кучи.

Определяем функцию пути, которая будет возвращать значение маршрута от начального до конечного узла.

Логикф процесса:

Добавляем начальную точку в «список посещения». Задаём стоп-условия, чтобы избежать бесконечного цикла. Задаём правила перемещения по позициям.

Нижеследующее повторяется, пока не сработает стоп-условие:

Выбираем клетку из «списка посещения» с минимальным показателем f. Эта позиция становится текущей. Проверяем, было ли достигнуто максимальное число итераций.

Проверяем, соответствует ли текущая клетка конечному квадрату-цели (то есть был ли определен маршрут).

Проверяем четыре клетки, прилегающих к текущей позиции. Если это тупик или уже посещенные клетки, игнорируем их. В противном случае создаём новый узел, для которого в качестве узла-родителя принимаем текущую клетку и обновляем позицию дочернего узла.

Проверяем все созданные дочерние узлы:

- если его нет в «списке посещения», то добавляем. Текущий узел становится узлом-родителем. Записываем стоимость f, g и h.

- если узел уже есть в списке, проверяем, будет ли путь через него лучше по показателю g (меняем родительский квадрат на текущий и пересчитываем g и f).

Для всех дочерних узлов:

а) если дочерний элемент уже находится в «списке посещения», игнорируем его и переходим к следующему;

б) вычисляем значения дочерних узлов g, h и f. Стоимость h для текущего узла вычисляется с использованием евклидова расстояния;

в) если дочерний узел уже в «списке посещения», игнорируем его. В противном случае добавляем дочерний узел в список.

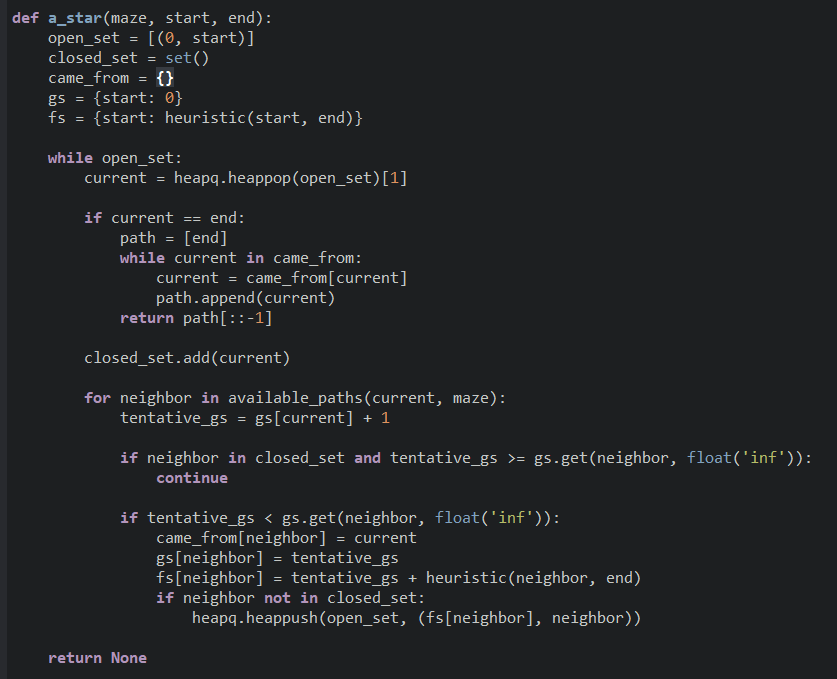
****

рис. 3.3. A\*.

# Пример работы

Программа находит путь, и, если работает без ошибок, сообщает об этом. Результат работы представляет собой лабиринт с символами «.»- траектория пути алгоритма поиска в глубину, «.»-траектория пути алгоритма А\*. Для наглядности продемонстрирую результат работы программы.

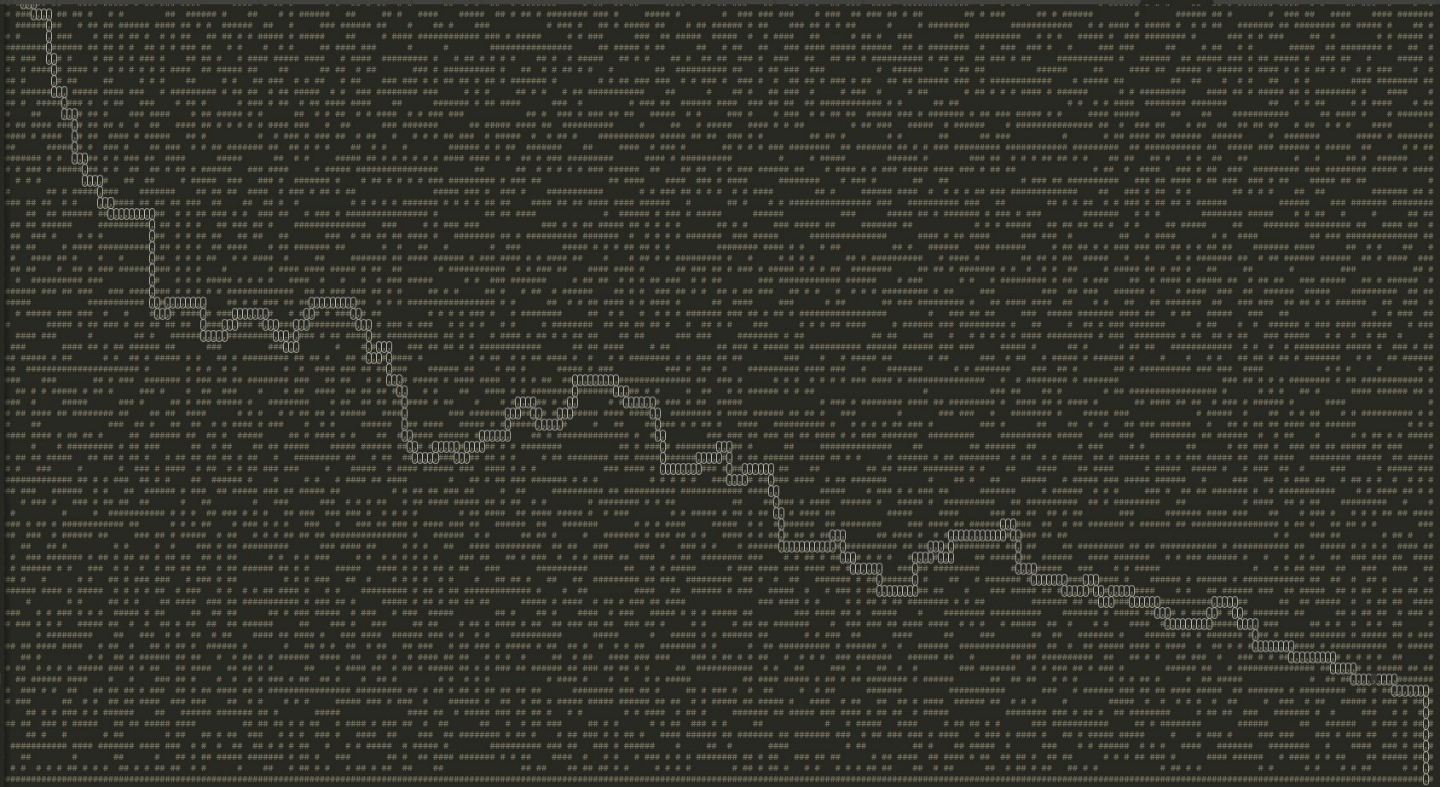


рис. 4.1. Результат.

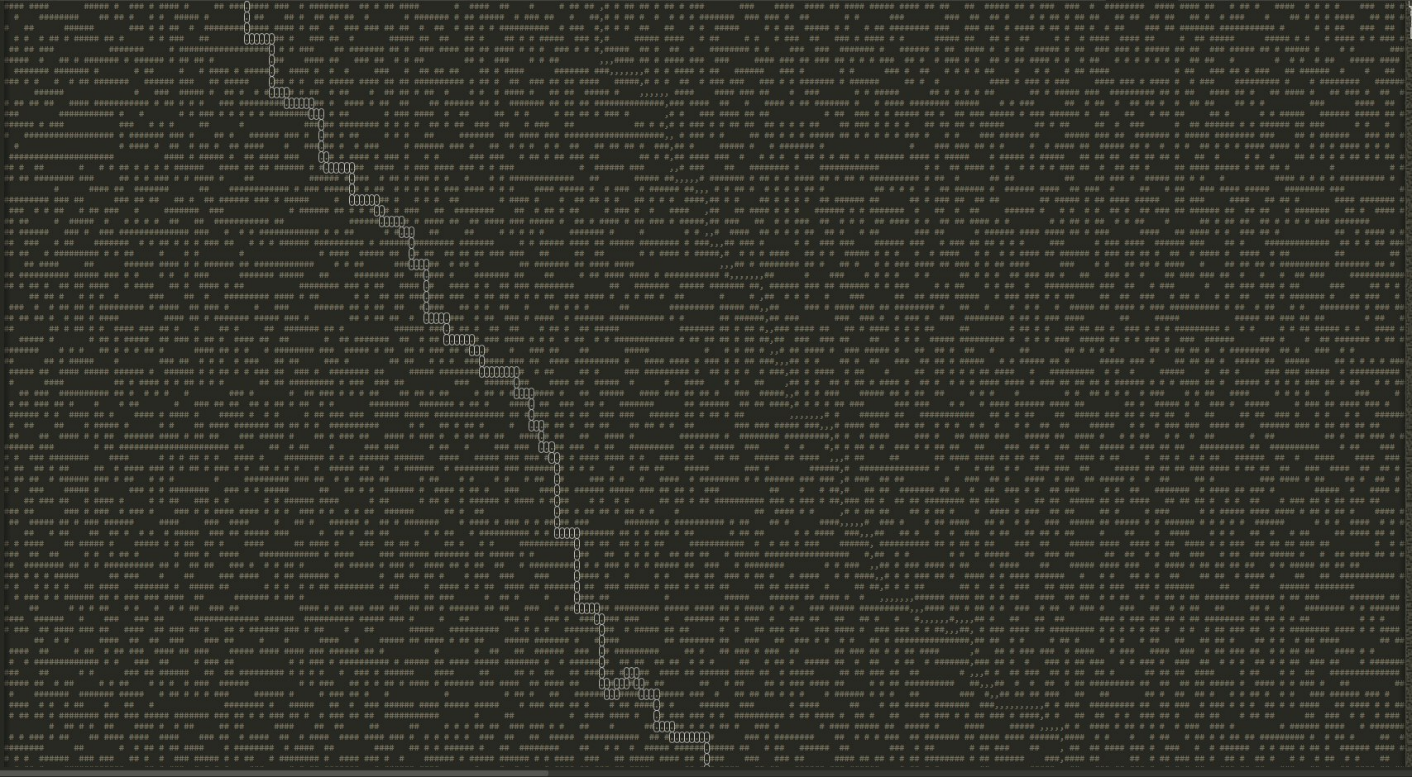


рис. 4.2. Результат.

# Заключение

В рамках выполнения данной работы были изучены теоретические основы алгоритмов обхода графа в глубину и А\*, структурного программирования.

Написана программа на языке Python, реализующая алгоритмы обхода графа в глубину и А\* для поиска маршрута в лабиринте. Результаты обходов лабиринта и получившиеся маршруты были сохранены в файл.

Структура программы состоит из модулей для работы с лабиринтом, а также модулей для реализации алгоритмов обхода в ширину и А\*.

Таким образом, цель курсовой работы достигнута. Результатом является программа, способная находить маршрут в лабиринте с помощью алгоритмов обхода графа в глубину и А\*, с сохранением полученных маршрутов в файле. Работа программы была проверена на нескольких тестовых лабиринтах и показала хорошие результаты.

# Список литературы

1. 10 Графовых алгоритмов URL: <https://medium.com/nuances-of-programming/10-графовых-алгоритмов-наглядное-объяснение-53226d8e6b0> (Дата обращения: 11.04.2023).
2. Агапов И., «Обход графа: поиск в глубину и поиск в ширину простыми словами на примере JavaScript» [Электронный ресурс]. URL: <https://habr.com/ru/articles/504374/> (Дата обращения: 11.04.2023).
3. Станковец, А.В. АЛГОРИТМЫ НА ГРАФАХ / Станковец А.В. – Текст: электронный // M[ODERN SCIENCE](https://www.elibrary.ru/contents.asp?id=44150018) ,2020, С.532-536.
4. A \* алгоритм поиска кратчайшего пути - Русские блоги URL: <https://russianblogs.com/article/48281474777/> (Дата обращения: 11.04.2023).
5. Применение рандомизированного поиска [в глубину (DFS)](https://translated.turbopages.org/proxy_u/en-ru.ru.20352f50-64518231-b8e12648-74722d776562/https/www.baeldung.com/cs/depth-first-search-intro) к сетке. URL: <https://translated.turbopages.org/proxy_u/en-ru.ru.20352f50-64518231-b8e12648-74722d776562/https/www.baeldung.com/cs/maze-generation> (Дата обращения: 11.04.2023).
6. Поиск в глубину. URL: <https://ru.wikipedia.org/wiki/Поиск_в_глубину> (Дата обращения: 11.04.2023).
7. Скиена С. Стивен, Алгоритмы. Руководство по разработке. 3-е изд, 2022, 848с.
8. DFS (Depth-First Search) - поиск в глубину: алгоритм обхода графа URL: <https://blog.skillfactory.ru/glossary/dfs/> (Дата обращения: 11.04.2023).
9. A\*. URL: [https://translated.turbopages.org/proxy\_u/en-ru.ru.47b4bdef-64524279-123590ef-74722d776562/https/en.wikipedia.org/wiki/A\*\_search\_algorithm](https://translated.turbopages.org/proxy_u/en-ru.ru.47b4bdef-64524279-123590ef-74722d776562/https/en.wikipedia.org/wiki/A*_search_algorithm) (Дата обращения: 11.04.2023).

# Приложение 1

## Листинг программы

## import heapq

## def heuristic(point, goal):

## return abs(point[0] - goal[0]) + abs(point[1] - goal[1])

## 

## def available\_paths(position, grid):

## available = []

## for delta in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

## new\_position = (position[0] + delta[0], position[1] + delta[1])

## if 0 <= new\_position[0] < len(grid) and 0 <= new\_position[1] < len(grid[0]) and grid[new\_position[0]][new\_position[1]] != '#':

## available.append(new\_position)

## return available

## 

## def a\_star(maze, start, end):

## open\_set = [(0, start)]

## closed\_set = set()

## came\_from = {}

## gs = {start: 0}

## fs = {start: heuristic(start, end)}

## while open\_set:

## current = heapq.heappop(open\_set)[1]

## 

## if current == end:

## path = [end]

## while current in came\_from:

## current = came\_from[current]

## path.append(current)

## return path[::-1]

## closed\_set.add(current)

## for neighbor in available\_paths(current, maze):

## tentative\_gs = gs[current] + 1

## 

## if neighbor in closed\_set and tentative\_gs >= gs.get(neighbor, float('inf')):

## continue

## 

## if tentative\_gs < gs.get(neighbor, float('inf')):

## came\_from[neighbor] = current

## gs[neighbor] = tentative\_gs

## fs[neighbor] = tentative\_gs + heuristic(neighbor, end)

## if neighbor not in closed\_set:

## heapq.heappush(open\_set, (fs[neighbor], neighbor))

## return None

## def dfs(maze, start, target, stack):

## #stack будет хранить итоговый путь

## #так же с каждым шагом увеличивая или уменьшая k будем заполнять dfs\_graph номерами всевозможных шагов из каждой текущей клетки

## #По итогу в dfs\_graph будут все возможные обходы произведенные алгоритмом dfs

## visited = set()#список посещенных координат

## k=1

## current = (start[0], start[1])

## visited.add(current)

## dfs\_graph[current[0]][current[1]]=k

## 

## while current!=target:

## 

## row, col = current

## #Список всех возможных соседей

## neighbors = [

## (row - 1, col), (row + 1, col),

## (row, col - 1), (row, col + 1)

## ]

## #Проверка имеет ли текущая клетка свободных, непосещенных соседей

## if (neighbors[0] not in visited) and 0 <= neighbors[0][0] < len(maze) and 0 <= neighbors[0][1] < len(maze[0]) and maze[neighbors[0][0]][neighbors[0][1]]!=1:

## stack.append(current)

## current=neighbors[0]

## visited.add(current)

## k+=1

## dfs\_graph[current[0]][current[1]]=k

## 

## elif (neighbors[3] not in visited) and 0 <= neighbors[3][0] < len(maze) and 0 <= neighbors[3][1] < len(maze[0]) and maze[neighbors[3][0]][neighbors[3][1]]!=1:

## stack.append(current)

## current=neighbors[3]

## visited.add(current)

## k+=1

## dfs\_graph[current[0]][current[1]]=k

## 

## elif (neighbors[1] not in visited) and 0 <= neighbors[1][0] < len(maze) and 0 <= neighbors[1][1] < len(maze[0]) and maze[neighbors[1][0]][neighbors[1][1]]!=1:

## stack.append(current)

## current=neighbors[1]

## visited.add(current)

## k+=1

## dfs\_graph[current[0]][current[1]]=k

## elif (neighbors[2] not in visited) and 0 <= neighbors[2][0] < len(maze) and 0 <= neighbors[2][1] < len(maze[0]) and maze[neighbors[2][0]][neighbors[2][1]]!=1:

## stack.append(current)

## current=neighbors[2]

## visited.add(current)

## k+=1

## dfs\_graph[current[0]][current[1]]=k

## 

## elif stack:

## current=stack.pop()

## k-=1

## else:

## return False

## 

## stack.append(current)

## return True

## 

## def create\_temple(temple, size\_w, size\_h):

## for i in range(size\_h):

## temple.append([])

## for j in range(size\_w):

## temple[-1].append(0)

## def crawl\_graph(graph, size\_w, size\_h):

## for i in range(size\_h):

## for j in range(size\_w):

## if graph[i][j]==1:

## graph[i][j]='#'

## 

## elif graph[i][j]==0:

## graph[i][j]=' '

## 

## elif graph[i][j]==3:

## graph[i][j]='.'

## 

## elif graph[i][j]==4:

## graph[i][j]=','

## 

## maze\_graph[i][size\_w-1]+='\n'

## 

## def get\_coord(max\_x, max\_y, style):

## while True:

## while True:

## try:

## if style==1:

## coord\_y = int(input(f"Enter y coordinate of AVATAR coord, y < {max\_y}:"))

## elif style==2 :

## coord\_y = int(input(f"Enter y coordinate of EXIT coord, y < {max\_y}:"))

## else:

## coord\_y = int(input(f"Enter y coordinate of KEY coord, y < {max\_y}:"))

## if coord\_y>=max\_y:

## print("Out of range")

## else:

## break;

## except ValueError:

## print("Enter number")

## while True:

## try:

## if style==1:

## coord\_x = int(input(f"Enter x coordinate of Avatar coord, x < {max\_x}:"))

## elif style==2 :

## coord\_x = int(input(f"Enter x coordinate of EXIT coord, x < {max\_x}:"))

## else:

## coord\_x = int(input(f"Enter x coordinate of KEY coord, x < {max\_x}:"))

## if coord\_x>=max\_x:

## print("Out of range")

## else:

## break;

## except ValueError:

## print("Enter a number")

## if (maze\_raw[coord\_y][coord\_x] == ' '):

## return coord\_y, coord\_x

## else:

## print("Those are coordinates of a wall");

## file\_name='maze-for-u.txt'

## maze\_raw = open(file\_name).read().split('\n')[:]

## height = len(maze\_raw)

## width = len(maze\_raw[0])

## print("Height of Maze", height)

## print("Width of Maze", width)

## start = get\_coord(width, height, 1)

## end = get\_coord(width, height, 2)

## key = get\_coord(width, height, 3)

## maze\_graph = []

## create\_temple(maze\_graph, width, height)

## for i in range(height):

## for j in range(width):

## if maze\_raw[i][j] == '#':

## maze\_graph[i][j]=1;

## #Ищем любой путь от старта до ключа с помощью алгоритма dfs(поиск в глубину)

## coord\_path\_key=[]

## dfs\_graph=[]

## create\_temple(dfs\_graph, width, height)

## dfs(maze\_graph, start, key, coord\_path\_key)

## for i in range(len(coord\_path\_key)):

## x=coord\_path\_key[i][1]

## y=coord\_path\_key[i][0]

## maze\_graph[y][x]= 3

## 

## #Ищем оптимальный путь от ключа до выхода с помощью алгоритма A\*

## coord\_path\_end=[]

## coord\_path\_end=a\_star(maze\_raw, key, end)

## for i in range(len(coord\_path\_end)):

## x=coord\_path\_end[i][1]

## y=coord\_path\_end[i][0]

## maze\_graph[y][x]= 4

## #отрисовываем лабиринт с нашим проходом в изначальном виде

## maze\_graph[key[0]][key[1]]= '\*'#ставим на график \*-ключ

## crawl\_graph(maze\_graph, width, height)

## #Преобразуем граф из массива символов в массив строк для печати в txt файл

## string\_row=''

## result\_path\_graph=[]

## for i in range(height):

## for j in range(width):

## string\_row+=maze\_graph[i][j]

## result\_path\_graph.append(string\_row)

## string\_row=''

## #Печатаем пройденный граф в файл

## with open('maze\_path\_result.txt', 'w') as file2:

## file2.writelines(result\_path\_graph)

## print("End of the algorithm!\nSee the result in the file: maze\_path\_result.txt") return abs(point[0] - goal[0]) + abs(point[1] - goal[1])

## 

## def available\_paths(position, grid):

## available = []

## for delta in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

## new\_position = (position[0] + delta[0], position[1] + delta[1])

## if 0 <= new\_position[0] < len(grid) and 0 <= new\_position[1] < len(grid[0]) and grid[new\_position[0]][new\_position[1]] != '#':

## available.append(new\_position)

## return available

## 

## def a\_star(maze, start, end):

## open\_set = [(0, start)]

## closed\_set = set()

## came\_from = {}

## gs = {start: 0}

## fs = {start: heuristic(start, end)}

## while open\_set:

## current = heapq.heappop(open\_set)[1]

## 

## if current == end:

## path = [end]

## while current in came\_from:

## current = came\_from[current]

## path.append(current)

## return path[::-1]

## closed\_set.add(current)

## for neighbor in available\_paths(current, maze):

## tentative\_gs = gs[current] + 1

## 

## if neighbor in closed\_set and tentative\_gs >= gs.get(neighbor, float('inf')):

## continue

## 

## if tentative\_gs < gs.get(neighbor, float('inf')):

## came\_from[neighbor] = current

## gs[neighbor] = tentative\_gs

## fs[neighbor] = tentative\_gs + heuristic(neighbor, end)

## if neighbor not in closed\_set:

## heapq.heappush(open\_set, (fs[neighbor], neighbor))

## return None

## def dfs(maze, start, target, stack):

## #stack будет хранить итоговый путь

## #так же с каждым шагом увеличивая или уменьшая k будем заполнять dfs\_graph номерами всевозможных шагов из каждой текущей клетки

## #По итогу в dfs\_graph будут все возможные обходы произведенные алгоритмом dfs

## visited = set()#список посещенных координат

## k=1

## current = (start[0], start[1])

## visited.add(current)

## dfs\_graph[current[0]][current[1]]=k

## 

## while current!=target:

## 

## row, col = current

## #Список всех возможных соседей

## neighbors = [

## (row - 1, col), (row + 1, col),

## (row, col - 1), (row, col + 1)

## ]

## #Проверка имеет ли текущая клетка свободных, непосещенных соседей

## if (neighbors[0] not in visited) and 0 <= neighbors[0][0] < len(maze) and 0 <= neighbors[0][1] < len(maze[0]) and maze[neighbors[0][0]][neighbors[0][1]]!=1:

## stack.append(current)

## current=neighbors[0]

## visited.add(current)

## k+=1

## dfs\_graph[current[0]][current[1]]=k

## 

## elif (neighbors[3] not in visited) and 0 <= neighbors[3][0] < len(maze) and 0 <= neighbors[3][1] < len(maze[0]) and maze[neighbors[3][0]][neighbors[3][1]]!=1:

## stack.append(current)

## current=neighbors[3]

## visited.add(current)

## k+=1

## dfs\_graph[current[0]][current[1]]=k

## 

## elif (neighbors[1] not in visited) and 0 <= neighbors[1][0] < len(maze) and 0 <= neighbors[1][1] < len(maze[0]) and maze[neighbors[1][0]][neighbors[1][1]]!=1:

## stack.append(current)

## current=neighbors[1]

## visited.add(current)

## k+=1

## dfs\_graph[current[0]][current[1]]=k

## elif (neighbors[2] not in visited) and 0 <= neighbors[2][0] < len(maze) and 0 <= neighbors[2][1] < len(maze[0]) and maze[neighbors[2][0]][neighbors[2][1]]!=1:

## stack.append(current)

## current=neighbors[2]

## visited.add(current)

## k+=1

## dfs\_graph[current[0]][current[1]]=k

## 

## elif stack:

## current=stack.pop()

## k-=1

## else:

## return False

## 

## stack.append(current)

## return True

## 

## def create\_temple(temple, size\_w, size\_h):

## for i in range(size\_h):

## temple.append([])

## for j in range(size\_w):

## temple[-1].append(0)

## def crawl\_graph(graph, size\_w, size\_h):

## for i in range(size\_h):

## for j in range(size\_w):

## if graph[i][j]==1:

## graph[i][j]='#'

## 

## elif graph[i][j]==0:

## graph[i][j]=' '

## 

## elif graph[i][j]==3:

## graph[i][j]='.'

## 

## elif graph[i][j]==4:

## graph[i][j]=','

## 

## maze\_graph[i][size\_w-1]+='\n'

## 

## def get\_coord(max\_x, max\_y, style):

## while True:

## while True:

## try:

## if style==1:

## coord\_y = int(input(f"Enter y coordinate of AVATAR coord, y < {max\_y}:"))

## elif style==2 :

## coord\_y = int(input(f"Enter y coordinate of EXIT coord, y < {max\_y}:"))

## else:

## coord\_y = int(input(f"Enter y coordinate of KEY coord, y < {max\_y}:"))

## if coord\_y>=max\_y:

## print("Out of range")

## else:

## break;

## except ValueError:

## print("Enter number")

## while True:

## try:

## if style==1:

## coord\_x = int(input(f"Enter x coordinate of Avatar coord, x < {max\_x}:"))

## elif style==2 :

## coord\_x = int(input(f"Enter x coordinate of EXIT coord, x < {max\_x}:"))

## else:

## coord\_x = int(input(f"Enter x coordinate of KEY coord, x < {max\_x}:"))

## if coord\_x>=max\_x:

## print("Out of range")

## else:

## break;

## except ValueError:

## print("Enter a number")

## if (maze\_raw[coord\_y][coord\_x] == ' '):

## return coord\_y, coord\_x

## else:

## print("Those are coordinates of a wall");

## file\_name='maze-for-u.txt'

## maze\_raw = open(file\_name).read().split('\n')[:]

## height = len(maze\_raw)

## width = len(maze\_raw[0])

## print("Height of Maze", height)

## print("Width of Maze", width)

## start = get\_coord(width, height, 1)

## end = get\_coord(width, height, 2)

## key = get\_coord(width, height, 3)

## maze\_graph = []

## create\_temple(maze\_graph, width, height)

## for i in range(height):

## for j in range(width):

## if maze\_raw[i][j] == '#':

## maze\_graph[i][j]=1;

## #Ищем любой путь от старта до ключа с помощью алгоритма dfs(поиск в глубину)

## coord\_path\_key=[]

## dfs\_graph=[]

## create\_temple(dfs\_graph, width, height)

## dfs(maze\_graph, start, key, coord\_path\_key)

## for i in range(len(coord\_path\_key)):

## x=coord\_path\_key[i][1]

## y=coord\_path\_key[i][0]

## maze\_graph[y][x]= 3

## 

## #Ищем оптимальный путь от ключа до выхода с помощью алгоритма A\*

## coord\_path\_end=[]

## coord\_path\_end=a\_star(maze\_raw, key, end)

## for i in range(len(coord\_path\_end)):

## x=coord\_path\_end[i][1]

## y=coord\_path\_end[i][0]

## maze\_graph[y][x]= 4

## #отрисовываем лабиринт с нашим проходом в изначальном виде

## maze\_graph[key[0]][key[1]]= '\*'#ставим на график \*-ключ

## crawl\_graph(maze\_graph, width, height)

## #Преобразуем граф из массива символов в массив строк для печати в txt файл

## string\_row=''

## result\_path\_graph=[]

## for i in range(height):

## for j in range(width):

## string\_row+=maze\_graph[i][j]

## result\_path\_graph.append(string\_row)

## string\_row=''

## #Печатаем пройденный граф в файл

## with open('maze\_path\_result.txt', 'w') as file2:

## file2.writelines(result\_path\_graph)

## print("End of the algorithm!\nSee the result in the file: maze\_path\_result.txt")