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 Студентов С.С.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc1)

[Задачи 3](#_Toc2)

[1.Теоретическая часть 4](#_Toc3)

[2. Реализация алгоритма 5](#_Toc4)

[Пример работы 6](#_Toc5)

[Заключение 7](#_Toc6)

[Список литературы 8](#_Toc7)

[Приложение 1 9](#_Toc8)

[Листинг программы 9](#_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. []

Структурное программирование — это технология создания программ, которая предполагает использование только трех основных структур управления: последовательности, ветвления и цикла. Таким образом, цель структурного программирования - повышение надежности программ, обеспечение сопровождения и модификации, облегчение и ускорение разработки. Структурное программирование широко используется в крупномасштабных проектах, когда на первый план выходят следующие преимущества: легкость повторного использования фрагментов кода, оформленных как процедуры (например, в библиотеках); легкость прослеживания логики программы; возможность сопровождения программного продукта через длительное время после написания кода или кем-то, кроме его автора. []

В рамках работы будут рассмотрены два ключевых алгоритма: поиск в глубину (DFS, или Depth First Search) и алгоритм A\* (A-star). Оба алгоритма являются эффективными инструментами для нахождения оптимальных путей в графах с различными характеристиками.

**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;

# 1.Теоретическая часть

* 1. **Поиск в глубину (DFS, или Depth First Search)**

DFS, или Depth First Search, — поиск в глубину, позволяющий найти маршрут от точки A до точки B. Используется в графах — особых структурах, состоящих из точек-вершин и ребер-путей. DFS ищет маршрут по графу “в глубину”: на каждом шаге “уходит” все дальше.

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.

Обход в глубину можно представить себе следующим образом. Пусть исследователь находится в некотором лабиринте (графе) и он хочет обойти весь лабиринт (посетить все доступные вершины в графе). Исследователь находится в некоторой вершине и видит ребра, исходящие из этой вершины. Очевидная последовательность действий исследователя такая:

1. Пойти в какую-нибудь смежную вершину.
2. Обойти все, что доступно из этой вершины.
3. Вернуться в начальную вершину.
4. Повторить алгоритм для всех остальных вершин, смежных из начальной.

Видим, что алгоритм является рекурсивным — для обхода всего графа нужно переместиться в соседнюю вершину, после чего повторить для этой вершины алгоритм обхода. Но возникает проблема зацикливания — если из вершины A можно перейти в вершину B, то из вершины B можно перейти в вершину A и рекурсия будет бесконечной. Для борьбы с рекурсией нужно применить очень простую идею — исследователь не должен идти в ту вершину, в которой он уже был раньше, то есть которая не представляет для него интерес (считаем, что интерес для исследователя представляют только вершины, в которых он не был ранее). Итак, уточненный алгоритм может выглядеть следующим образом:

1. Пойти в какую-нибудь смежную вершину, не посещенную ранее.
2. Запустить из этой вершины алгоритм обхода в глубину
3. Вернуться в начальную вершину.
4. Повторить пункты 1-3 для всех не посещенных ранее смежных вершин.

Кто пользуется алгоритмом:

* Математики, которые работают с теорией графов для решения фундаментальных или практических задач.
* Специалисты по анализу данных и по искусственному интеллекту, так как графы часто используются в Data Science или в машинном обучении.
* Разработчики, которым бывает необходимо решать задачи поиска маршрутов, расчета потоков и другие подобные. Такие задачи могут встретиться во многих проектах: от картографического сервиса до онлайн-игры.
* Сетевые инженеры, так как в виде графов представляют компьютерные сети, а многие сетевые протоколы основаны на алгоритмах работы с графами.
* Иногда — другие специалисты, которым бывает нужно столкнуться с теорией графов. Вариации DFS используются в том числе в жизни.

Для чего нужен алгоритм DFS:

* Для поиска любого маршрута в лабиринте.
* 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.
* Как составная часть расчетов в более сложных алгоритмах, например, для определения максимального транспортного потока.
* Для решения ряда задач из теории графов, которые используются в программировании и математике: поиска циклов, сортировки и так далее.

Временная сложность алгоритма BFS составляет O (V + E), где V - количество вершин, а E - количество ребер в графе. Это обеспечивает достаточно эффективную работу алгоритма даже для больших графов.

**1.2Алгоритм А\*. Поиск кратчайшего пути.**

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.

Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как f(x)). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x) и может быть, как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

Функция h(x) должна быть допустимой эвристической оценкой, то есть не должна переоценивать расстояния к целевой вершине. Например, для задачи маршрутизации h(x) может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.

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)

f(n): общая стоимость пути

g(n): стоимость пути между текущей и начальной вершиной

h(n): эвристическая функция

[Временна́я сложность](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0) алгоритма A\* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт [экспоненциально](https://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D1%81%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C) по сравнению с длиной оптимального пути, но сложность становится [полиномиальной](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BD%D0%BE%D0%BC%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C), когда пространство поиска является деревом, а эвристика удовлетворяет следующему условию:

|h(x)-h\*(x)|<=O(logh\*(x));

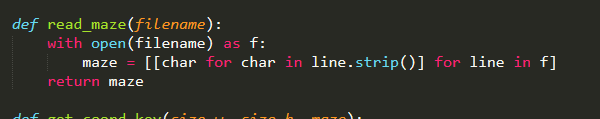
где h\* — оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка h(x) не должна расти быстрее, чем [логарифм](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B0%D1%80%D0%B8%D1%84%D0%BC) от оптимальной эвристики.

**Итоги**

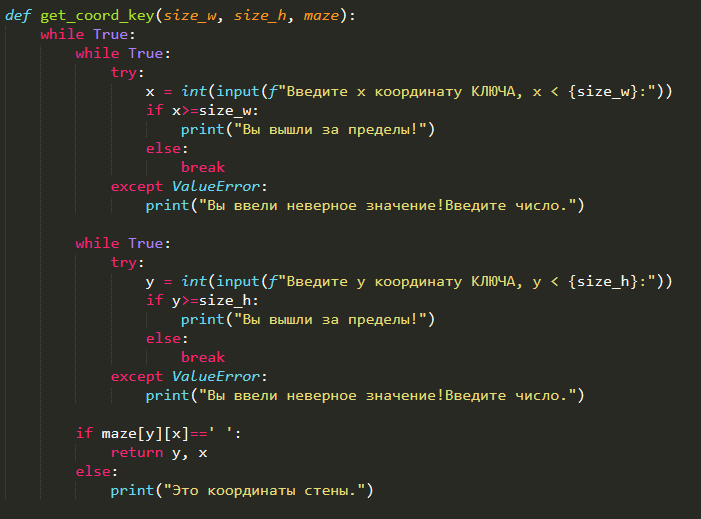
Алгоритм А\* — один из самых часто используемых алгоритмов поиска кратчайшего пути. Сегодня этот алгоритм применяют в различных областях от машинного обучения до разработки игр, например, при выстраивании маршрута по сложной местности с препятствиями.

# 2. Реализация алгоритма

Для начала создадим функцию read\_maze, которая будет читать с файла исходный лабиринт. Функция вернет двумерный массив символов.

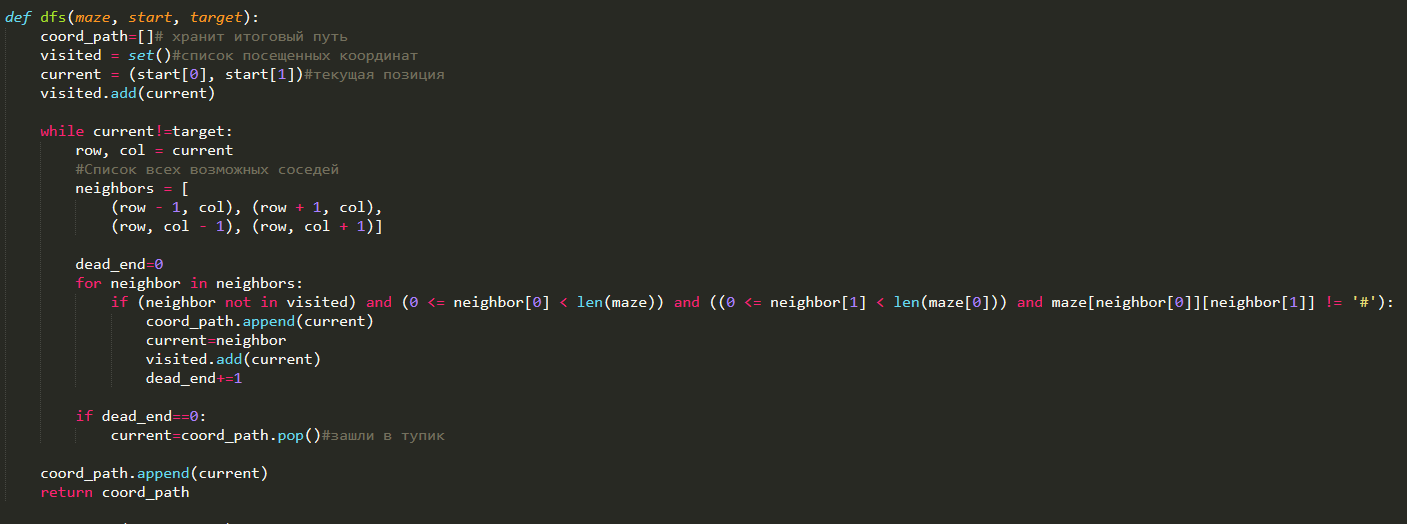


Затем создаю функцию get\_coord\_key, для считывания координат ключа. Эта функция будет проверять корректность введенных данных и возвратит координаты нахождения ключа.



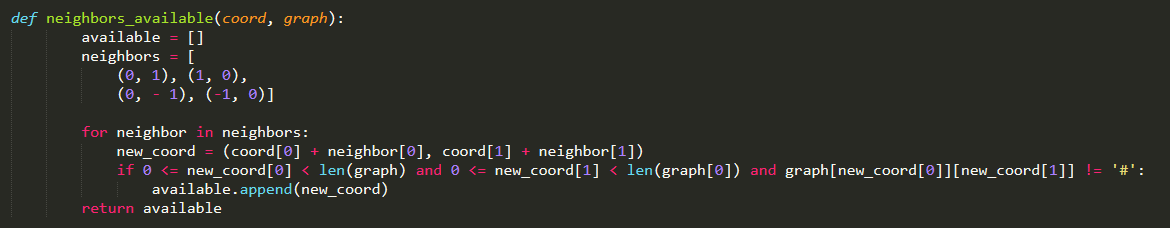
Далее приступаю к реализации алгоритма dfs. Coord\_path хранит в себе итоговой путь из кортежей координат. Visited список посещенных координат которые мы уже исследовали. Current это координата текущей позиции.

И так, пока текущая позиция не равна конечной позиции, мы сначала будем искать соседей, то есть клетки, которые мы еще не посещали и которые не выходят за размеры лабиринта и также не являются стенами. Если такие есть, то мы будем их добавлять в наш «стек», и перемещать положение текущего, а также добавлять это координату в список посещенных. Если у клетки нету соседей, удовлетворяющих этим условиям, то мы будем убирать из стека последнюю добавленную координату и повторять для нее вышеописанные действия. По итогу мы получим итоговой путь из кортежей координат.

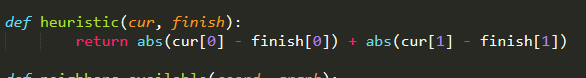


Для реализации алгоритма A\* потребуется дополнительные функции.

Функция neighbors\_available(coord, graph) находит все координаты доступных соседей, которые не выходят за размеры лабиринта и также не являются стенами. И возвращает их координаты.



Функция heuristic(cur, finish) - эвристическое расстояние от соседней точки до конечной точки. Общая стоимость нового пути вычисляется как сумма фактической стоимости пути и эвристического расстояния.



Функция a\_star реализует алгоритм A\*. Алгоритм начинается с пустого списка result\_path, visited, словаря distance и словаря previous. Текущая точка добавляется в distance со значением 0, а затем добавляется в open\_set. Пока есть элементы в open\_set, текущая точка current удаляется из него, а также добавляется в visited-посещенные точки. Если current - это конечная точка, то путь до этой точки был найден. В противном случае находятся соседние точки, и для каждой из них вычисляется расстояние до current и обновляются значения в словаре distance и previous. Если соседняя точка еще не находится в distances, то она добавляется в open\_set. В конце, если конечная точка находится в словаре previous, алгоритм проходит по списку previous, начиная с конечной точки и до начальной точки, добавляя каждую точку в path. Это продолжится до тех пор, пока конечная точка не будет достигнута или пока open\_set не опустеет. Если конечная точка не может быть достигнута из начальной точки, функция вернет None. Если конечная точка будет достигнута алгоритм вернет путь path.

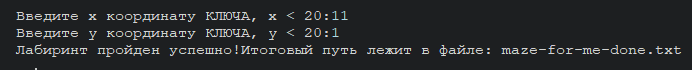


В функции main вызываются все вышеперечисленные функции. Сначала read\_maze считывает исходный лабиринт. Затем задаются координаты старта и конца, а также считываются координаты ключевой точки. Алгоритм поиска в глубину выполняется от начальной точки до ключевой точки, A\* выполняется от ключевой точки до конечной точки. Затем путь, найденный поиском в глубину, помечается символом ".", а путь, найденный A\*, помечается символом ",". Ключевая точка помечается символом "\*". Далее двумерный массив символов преобразовывается в массив строк, которым в конец добавляется символ перехода на новую строку "\n". Новый лабиринт записывается в файл `maze-for-me-done.txt`. В конце функция выводит уведомление о завершении поиска пути и местоположении нового лабиринта.

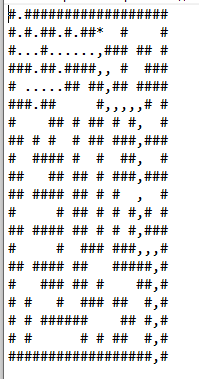
# Пример работы

1.Тестовый лабиринт

Входные данные:

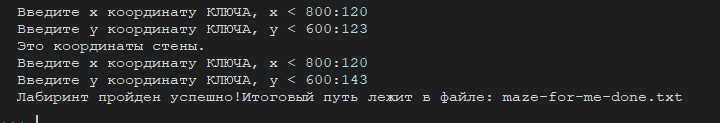


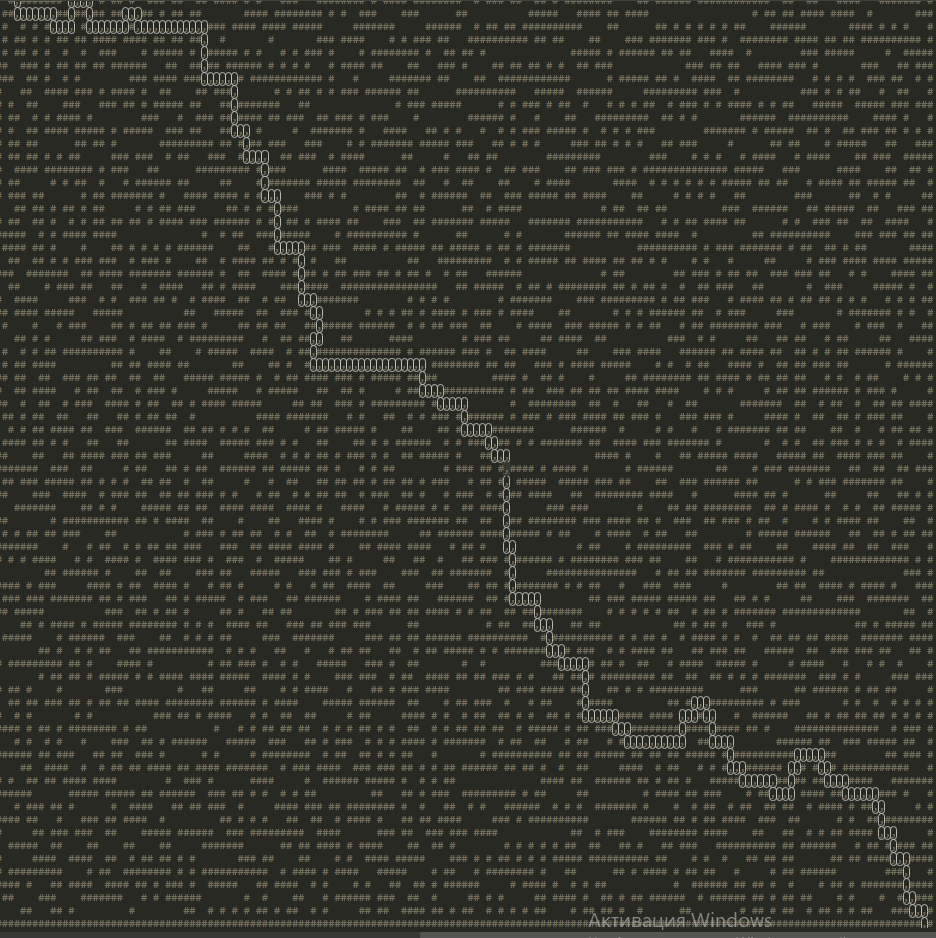
Результат:



Тест 2. Сгенерированный лабиринт.

Входные данные:





# Заключение

В ходе выполнения данной задачи были изучены теоретические основы алгоритмов прохода по графу в глубину и А\*, а также 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\*. Результатом является программа, способная находить оптимальный путь в лабиринте с использованием алгоритмов прохода по графу в ширину и А\*, а также сохранять полученные пути в файле.

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. Лекция 10. Структурное программирование, предпрограммная подготовка задачи// StudFiles / [Электронный ресурс]. URL: <https://studfile.net/preview/4290050/> (Дата обращения: 14.05.2023)
2. Кормен, Томас X. и др. «Алгоритмы: построение и анализ», 3-е изд.: Пер. с англ. – М.: ООО “И. Д. Вильямс” 2013. – 630 с.
3. Басараб М.А., Домрачева А.Б., Купляков В.М. Алгоритмы решения задачи быстрого поиска пути на географических картах. / Басараб М.А., Домрачева А.Б., Купляков В.М. - Текст: электронный // Инженерный журнал: наука и инновации. - 2013. - № 11. – с. 8 .
4. Авачева Т. Г., Пруцков А. В. Современный взгляд на концепцию структурного программирования (рус.) // Cloud of Science : Журнал. — 2019. — Т. 6, № 4. — С. 646–665.
5. 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/> .
6. Алгоритм DFS. <https://blog.skillfactory.ru/glossary/dfs/>
7. Алгоритм А\*. [https://translated.turbopages.org/proxy\_u/en-ru.ru.bbc2706e-64739e0a-dfad6bad-74722d776562/https/en.wikipedia.org/wiki/A\*\_search\_algorithm](https://translated.turbopages.org/proxy_u/en-ru.ru.bbc2706e-64739e0a-dfad6bad-74722d776562/https/en.wikipedia.org/wiki/A*_search_algorithm)
8. Алгоритмы обхода графа. <https://medium.com/nuances-of-programming/10-графовых-алгоритмов-наглядное-объяснение-53226d8e6ba0>
9. A \* алгоритм поиска кратчайшего пути. Эта статья переведена из блога Патрика Лестера. <https://russianblogs.com/article/48281474777/>

# Приложение 1

## Листинг программы

## import heapq

## def read\_maze(filename):

## with open(filename) as f:

## maze = [[char for char in line.strip()] for line in f]

## return maze

## def get\_coord\_key(size\_w, size\_h, maze):

## while True:

## while True:

## try:

## x = int(input(f"Введите x координату КЛЮЧА, x < {size\_w}:"))

## if x>=size\_w:

## print("Вы вышли за пределы!")

## else:

## break

## except ValueError:

## print("Вы ввели неверное значение!Введите число.")

## 

## while True:

## try:

## y = int(input(f"Введите y координату КЛЮЧА, y < {size\_h}:"))

## if y>=size\_h:

## print("Вы вышли за пределы!")

## else:

## break

## except ValueError:

## print("Вы ввели неверное значение!Введите число.")

## 

## if maze[y][x]==' ':

## return y, x

## else:

## print("Это координаты стены.")

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

## coord\_path=[]# хранит итоговый путь

## visited = set()#список посещенных координат

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

## visited.add(current)

## 

## while current!=target:

## row, col = current

## #Список всех возможных соседей

## neighbors = [

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

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

## dead\_end=0

## for neighbor in neighbors:

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

## coord\_path.append(current)

## current=neighbor

## visited.add(current)

## dead\_end+=1

## 

## if dead\_end==0:

## current=coord\_path.pop()#зашли в тупик

## 

## coord\_path.append(current)

## return coord\_path

## def heuristic(cur, finish):

## return abs(cur[0] - finish[0]) + abs(cur[1] - finish[1])

## def neighbors\_available(coord, graph):

## available = []

## neighbors = [

## (0, 1), (1, 0),

## (0, - 1), (-1, 0)]

## 

## for neighbor in neighbors:

## new\_coord = (coord[0] + neighbor[0], coord[1] + neighbor[1])

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

## available.append(new\_coord)

## return available

## 

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

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

## visited = set()

## result\_path = {}

## distance = {start: 0}

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

## while open\_set:

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

## 

## if current == end:

## path = [end]

## while current in result\_path:

## current = result\_path[current]

## path.append(current)

## return path[::-1]

## visited.add(current)

## for neighbor in neighbors\_available(current, maze):

## tentative\_distance = distance[current] + 1

## 

## if neighbor in visited and tentative\_distance >= distance.get(neighbor, float('inf')):

## continue

## 

## if tentative\_distance < distance.get(neighbor, float('inf')):

## result\_path[neighbor] = current

## distance[neighbor] = tentative\_distance

## previous[neighbor] = tentative\_distance + heuristic(neighbor, end)

## if neighbor not in visited:

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

## return None

## def main():

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

## init\_maze = read\_maze(file\_name)

## start = (0, 1)

## end = (len(init\_maze)-1, len(init\_maze[0])-2)

## key = get\_coord\_key(len(init\_maze[0]), len(init\_maze), init\_maze)

## key\_path=dfs(init\_maze, start, key)

## end\_path=a\_star(init\_maze, key, end)

## 

## for coord in key\_path:

## init\_maze[coord[0]][coord[1]]='.'

## 

## for coord2 in end\_path:

## init\_maze[coord2[0]][coord2[1]]=','

## 

## init\_maze[key[0]][key[1]]='\*'

## result\_maze=[]

## for line in init\_maze:

## result\_line=""

## for symbol in line:

## result\_line+=symbol

## result\_line+='\n'

## result\_maze.append(result\_line)

## 

## with open('maze-for-me-done.txt', 'w') as file2:

## file2.writelines(result\_maze)

## print("Лабиринт пройден успешно!Итоговый путь лежит в файле: maze-for-me-done.txt")

## main()