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 Сорокина Д.А.

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 17](#_Toc6)

[Список литературы 18](#_Toc7)

[Приложение 1 19](#_Toc8)

[Листинг программы 19](#_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. [7] 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). [7]

**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).**

Алгоритм поиска в глубину стартует с определенной вершины и продолжает движение вдоль каждой ветви, возвратившись к предыдущей вершине при необходимости. Поиск в глубину требует отслеживания посещенных вершин и использует структуру данных "стек", чтобы обеспечить возврат к предыдущей вершине. Классический пример использования алгоритма - поиск случайного пути в лабиринте. Алгоритм начинает работу в заданной точке, на каждом шаге выбирает следующее направление и продвигается вперед по лабиринту. Если путь становится тупиковым, алгоритм возвращается к предыдущей точке и пытается другое направление. Это продолжается, пока не будет найден путь к целевой вершине. Поиск в глубину часто используется для нахождения пути между двумя вершинами и обнаружения циклов на графе. Алгоритм также применяется в топологической сортировке и головоломках с единственным решением, например, лабиринтах. Задача поиска выхода из лабиринта известна давно. В терминах графов лабиринт представляет неориентированный граф, где вершины представляют перекрестки, а ребра - дорожки между соседними перекрестками. Выходы обозначаются особыми вершинами. Задача состоит в нахождении пути от исходной вершины к вершине-выходу.

Мы рассмотрим метод обхода всех вершин графа, известный как поиск в глубину. Его идея заключается в следующем:

Начиная с некоторой вершины v, перемещаемся в произвольную смежную еще не посещенную вершину w. Если такая вершина отсутствует, возвращаемся в вершину, из которой пришли в v.

Какие ресурсы требуются для работы алгоритма?

Алгоритм проходит каждое ребро единожды и выполняет константное число действий для каждой вершины. Если число вершин обозначить как W, а число ребер - как V, то время работы алгоритма составляет O(W+V).

Нерекурсивные варианты.

Рекурсивные реализации алгоритма требуют большого количества ресурсов, поэтому существуют несколько вариантов реализации без их использования.

Самый простой вариант — по-особому хранить в памяти посещенные вершины. Для этого используется такая структура данных, как стек. 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.**

В поиске кратчайшего маршрута ежедневно используют алгоритм А \*, один из наиболее известных методов решения подобных задач. Он относится к информированным алгоритмам поиска, так как использует данные о стоимости пути и принципы эвристики.

A \* (произносится как "A-star") - это алгоритм обхода графа, который применяется во многих областях информатики благодаря своей полноте, оптимальности и оптимальной эффективности. Его основным недостатком является высокая пространственная сложность O(b^d), так как он хранит все сгенерированные узлы в памяти. Однако для многих задач маршрутизации он по-прежнему остается лучшим решением.

Оптимальность и полнота - две ключевые характеристики алгоритмов такого рода. Оптимальный алгоритм гарантирует получение наилучшего из возможных решений, а полный алгоритм всегда находит решение, если таковое существует.

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\* является алгоритмом поиска с наилучшим первым выбором, который основан на обоснованном поиске. Этот поиск применим в случае взвешенных графов: алгоритм начинает работу с определенной начальной точки графа и ищет оптимальный путь до целевой точки с наименьшими затратами (минимальным расстоянием, временем и т.д.). Для этого он строит дерево путей, начинающееся в начальной точке, и расширяет эти пути, добавляя по одному ребру за раз, пока не будет достигнута целевая точка. На каждой итерации основного цикла A\* выбирает путь для расширения на основании его стоимости и оценки затрат, необходимых для продления этого пути до целевой точки. При посещении каждой точки подсчитывается ее стоимость f(n) (для n - смежной точки). Таким образом, алгоритм просматривает все соседние точки и выбирает точку с наименьшим значением этого показателя.

Формула выглядит следующим образом:

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

# 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\_blank\_temple создает двумерный массив нужного нам размера, заполненный нулями. Это понадобится нам для создания шаблонов под графы.

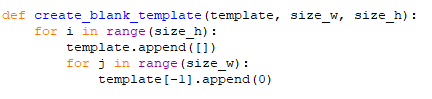


Рисунок 1. Функция create\_blank\_temple

* Функция input\_coord 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.



Рисунок 2. Функция input\_coord

* Функция completed\_maze 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).

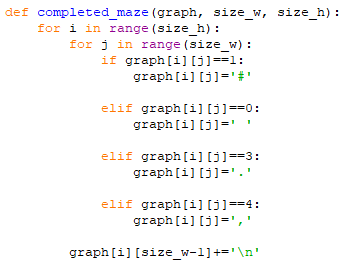


Рисунок 3. Функция completed\_maze

**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.

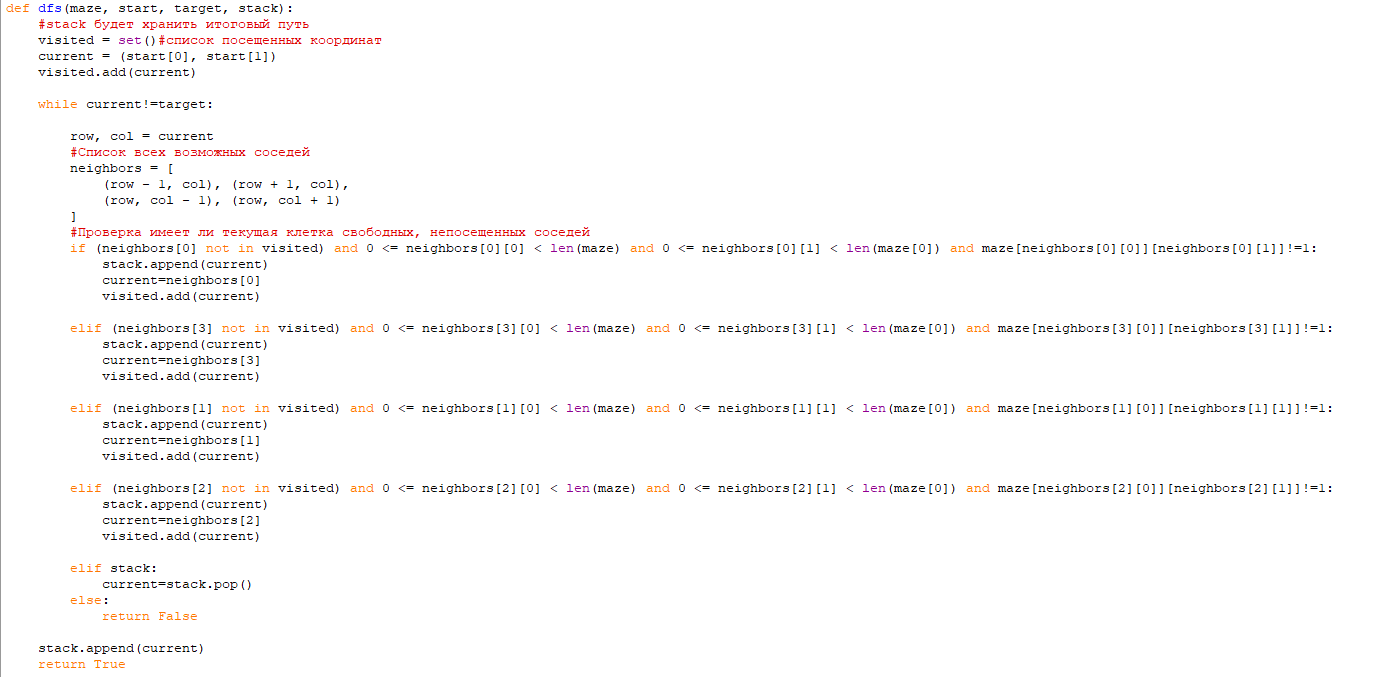


Рисунок 4. Реализация алгоритма поиска в глубину (Depth-First-Search, DFS).

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

**1 Этап**. Считываем лабиринт с файла и находим его размеры.

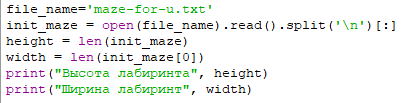


Рисунок 5. Считываем лабиринт и находим размеры

Просим пользователя ввести координаты и обрабатываем их.



Рисунок 6. Ввод и обработка координат

Создаем граф и преобразуем его текстовое представление в виде двумерного массива 0 и 1.

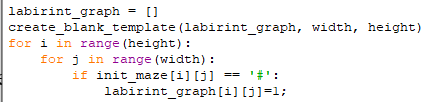


Рисунок 7. Создаём и преобразуем граф

**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.

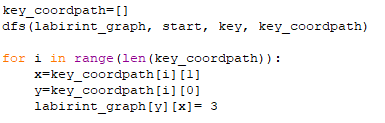


Рисунок 8. Ищем оптимальный путь от входа до ключа с помощью поиска в глубину

**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.

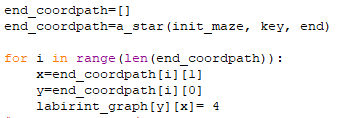


Рисунок 9. Ищем путь от ключа до выхода с помощью А\*

**4 Этап**. Граф готов. Меняем его представление для печати его в txt файл.

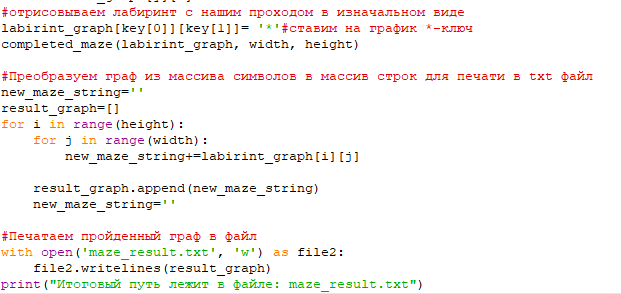


Рисунок 10. Меняем готовый граф для печати

**3. Реализация алгоритма A\*.**

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

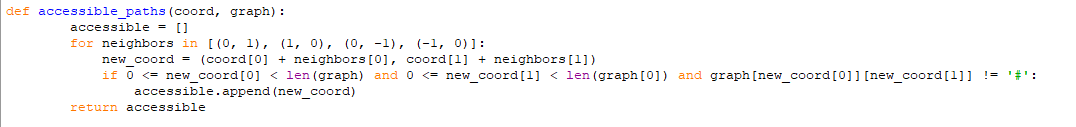
****

Рисунок 11. Вспомогательная функция accessible\_paths

Также понадобится эвристическая функция, сообщающая нам, насколько мы близки к цели.

****

Рисунок 12. heuristic

И наконец, сам алгоритм a\_star, 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.

Логика процесса:

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

До выполнения стоп-условия мы выполняем следующие действия:

Выбираем следующую клетку из «списка посещения», имеющую наименьший показатель f. Эта клетка становится текущей. Проверяем, не достигли ли мы максимального числа итераций. Проверяем, является ли текущая клетка конечной точкой (целью), для которой был найден маршрут. Разбираемся с четырьмя соседними клетками, примыкающими к текущей. Если какие-то клетки уже посещались или представляют собой тупик, игнорируем их. В противном случае создаем новый узел, который имеет текущую клетку в качестве узла-родителя, и обновляем позицию этого 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.

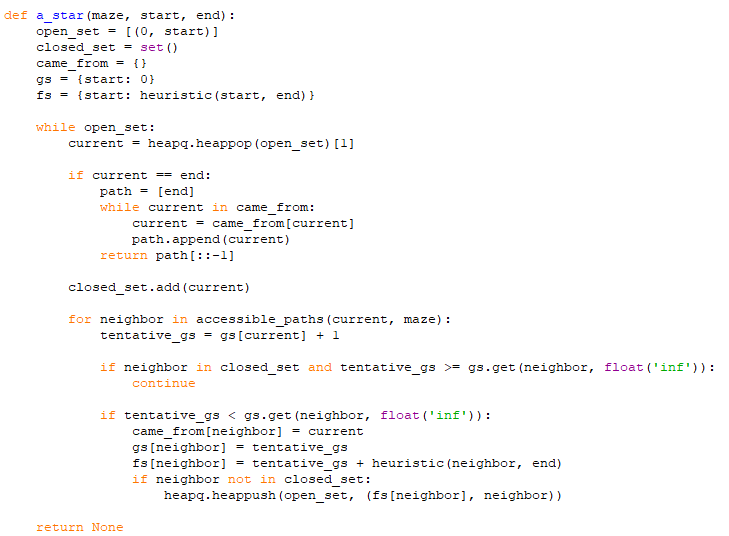
****

Рисунок 13. А\*

# Пример работы

Входной файл:

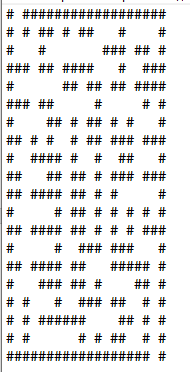


Рисунок 14. Часть входного файла

Введенные параметры:

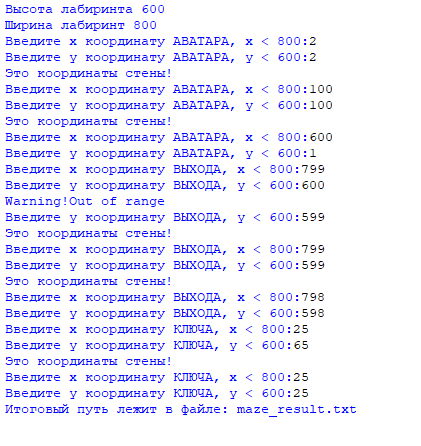


Рисунок 15. Введённые параметры

Полученный результат:

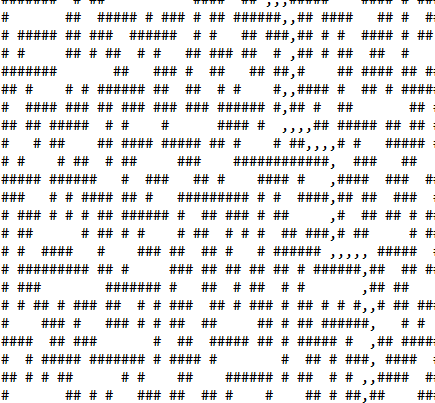


Рисунок 16. Результат

# Заключение

В процессе выполнения данной работы были углублено изучены теоретические принципы алгоритмов обхода графа в глубину и А\*, а также структурного программирования.

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.

Была разработана программа на языке Python, которая реализует оба алгоритма и используется для поиска маршрутов в лабиринте. Полученные маршруты сохраняются в файле. Структура программы включает модули для работы с лабиринтом и модули, реализующие алгоритмы обхода в глубину и А\*.

Цель работы достигнута - получили программу, способную найти маршрут в лабиринте, используя алгоритмы обхода графа в глубину и А\*, и сохранив полученные маршруты в файле. Работоспособность программы проверена и результаты оказались успешными.

# Список литературы

1. Скиена С. Стивен Алгоритмы. Руководство по разработке. 3-е изд / Скиена С. Стивен. – СПб.: Springer, 2022. – 848 с. – Текст:  
   непосредственный.
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. — №10-2. — С.532-536. — EDN: [pdmxpx](https://www.elibrary.ru/pdmxpx)
4. A \* алгоритм поиска кратчайшего пути. Эта статья переведена из блога Патрика Лестера. <https://russianblogs.com/article/48281474777/>
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. <https://translated.turbopages.org/proxy_u/en-ru.ru.20352f50-64518231-b8e12648-74722d776562/https/www.baeldung.com/cs/maze-generation>
6. Поиск в глубину. <https://ru.wikipedia.org/wiki/Поиск_в_глубину>
7. Алгоритмы обхода графа. <https://medium.com/nuances-of-programming/10-графовых-алгоритмов-наглядное-объяснение-53226d8e6ba0>
8. Про DFS. <https://blog.skillfactory.ru/glossary/dfs/>
9. A\*. https://translated.turbopages.org/proxy\_u/en-ru.ru.47b4bdef-64524279-123590ef-74722d776562/https/en.wikipedia.org/wiki/A\*\_search\_algorithm

# Приложение 1

## Листинг программы

import heapq

def input\_coord(max\_x, max\_y, tip):

while True:

while True:

try:

if tip==1:

coord\_x = int(input(f"Введите x координату АВАТАРА, x < {max\_x}:"))

elif tip==2 :

coord\_x = int(input(f"Введите x координату ВЫХОДА, x < {max\_x}:"))

else:

coord\_x = int(input(f"Введите x координату КЛЮЧА, x < {max\_x}:"))

if coord\_x>=max\_x:

print("Warning!Out of range")

else:

break;

except ValueError:

print("Ошибка!Введите число.")

while True:

try:

if tip==1:

coord\_y = int(input(f"Введите y координату АВАТАРА, y < {max\_y}:"))

elif tip==2 :

coord\_y = int(input(f"Введите y координату ВЫХОДА, y < {max\_y}:"))

else:

coord\_y = int(input(f"Введите y координату КЛЮЧА, y < {max\_y}:"))

if coord\_y>=max\_y:

print("Warning!Out of range")

else:

break;

except ValueError:

print("Ошибка!Введите число.")

if (init\_maze[coord\_y][coord\_x] == ' '):

return coord\_y, coord\_x

else:

print("Это координаты стены!");

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

#stack будет хранить итоговый путь

visited = set()#список посещенных координат

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

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)

]

#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)

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)

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)

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)

elif stack:

current=stack.pop()Ы

else:

return False

stack.append(current)

return True

def heuristic(point, goal):

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

def accessible\_paths(coord, graph):

accessible = []

for neighbors in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

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

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

accessible.append(new\_coord)

return accessible

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 accessible\_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 completed\_maze(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]=','

graph[i][size\_w-1]+='\n'

def create\_blank\_template(template, size\_w, size\_h):

for i in range(size\_h):

template.append([])

for j in range(size\_w):

template[-1].append(0)

#main

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

init\_maze = open(file\_name).read().split('\n')[:]

height = len(init\_maze)

width = len(init\_maze[0])

print("Высота лабиринта", height)

print("Ширина лабиринт", width)

start = input\_coord(width, height, 1)

end = input\_coord(width, height, 2)

key = input\_coord(width, height, 3)

labirint\_graph = []

create\_blank\_template(labirint\_graph, width, height)

for i in range(height):

for j in range(width):

if init\_maze[i][j] == '#':

labirint\_graph[i][j]=1;

key\_coordpath=[]

dfs(labirint\_graph, start, key, key\_coordpath)

for i in range(len(key\_coordpath)):

x=key\_coordpath[i][1]

y=key\_coordpath[i][0]

labirint\_graph[y][x]= 3

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

end\_coordpath=[]

end\_coordpath=a\_star(init\_maze, key, end)

for i in range(len(end\_coordpath)):

x=end\_coordpath[i][1]

y=end\_coordpath[i][0]

labirint\_graph[y][x]= 4

#отрисовываем лабиринт с нашим проходом в изначальном виде

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

completed\_maze(labirint\_graph, width, height)

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

new\_maze\_string=''

result\_graph=[]

for i in range(height):

for j in range(width):

new\_maze\_string+=labirint\_graph[i][j]

result\_graph.append(new\_maze\_string)

new\_maze\_string=''

#Печатаем пройденный граф в файл

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

file2.writelines(result\_graph)

print("Итоговый путь лежит в файле: maze\_result.txt")