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-34 Крылов К.Д.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc134907779)

[Задачи 3](#_Toc134907780)

[Теоретическая часть 4](#_Toc134907781)

[Реализация алгоритма 9](#_Toc134907782)

[Пример работы 10](#_Toc134907783)

[Заключение 11](#_Toc134907784)

[Список литературы 12](#_Toc134907785)

[Приложение 1 13](#_Toc134907786)

[Листинг программы 13](#_Toc134907787)

# **Введение**

Алгоритмы обхода графа являются одной из важнейших задач в программировании. Алгоритмы глубокого поиска пути являются одними из наиболее распространенных алгоритмов в области компьютерного зрения и машинного обучения. Они используются для решения задач, связанных с поиском оптимального пути в графах, а также для поиска объектов на изображениях и видео.

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

# **Задачи**

1. В папке \_3 запустить скрипт gen\_lab\_origin.py, подождать его работу несколько минут;
2. Полученный файл maze-for-u.txt переместить в свою папку;
3. Найти любой маршрут от начальной координаты аватара до ключа, используя жадный алгоритм.
4. Используя А\* со следующими параметрами (вес g(x), вес h(x), максимальная длина хранимого списка возможных шагов), найти оптимальный путь до ближайшего выхода;
5. Сохранить в файл 'maze-for-me-done.txt', в котором точка ключа будет указана как '\*', а сам маршрут построен точками к ключу и запятыми от него к выходу.

# **Теоретическая часть**

Алгоритмы глубокого поиска пути являются классом алгоритмов, которые используются для поиска оптимального пути в графе. Они могут быть использованы для решения различных задач, таких как поиск кратчайшего пути между двумя вершинами, поиск пути с минимальной стоимостью и т.д. Алгоритмы глубокого поиска пути широко применяются в задачах компьютерного зрения. Они используются для поиска объектов на изображениях и видео, а также для распознавания образов. Например, в задаче распознавания лиц алгоритм A\* может быть использован для поиска оптимального пути между точками, которые представляют глаза, нос и рот. Это может помочь улучшить точность распознавания. Один из наиболее распространенных алгоритмов глубокого поиска пути - это алгоритм A\*. Он основан на использовании эвристической функции, которая оценивает стоимость пути от начальной вершины до конечной. Эта функция используется для принятия решения о том, какой следующий узел должен быть исследован.

Стандартная реализация поиска в глубину помещает каждую вершину (узел, node) графа в одну из двух категорий:

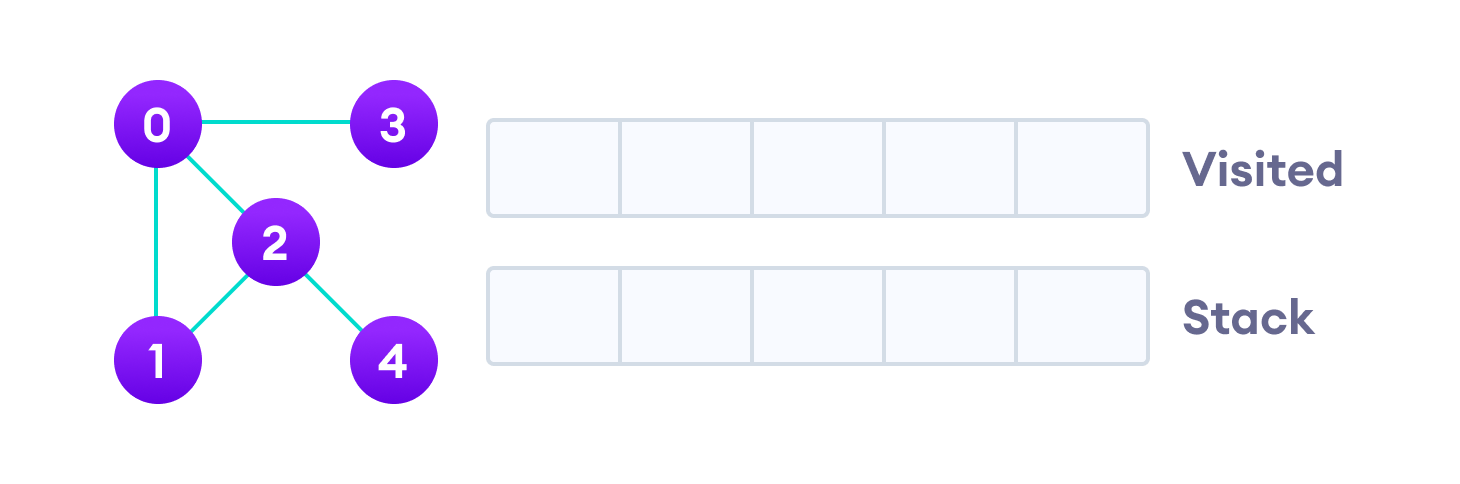
1. Пройденные (Visited).
2. Не пройденные (Not Visited).

Цель алгоритма состоит в том, чтобы пометить каждую вершину как “Пройденная”, избегая при этом циклов.

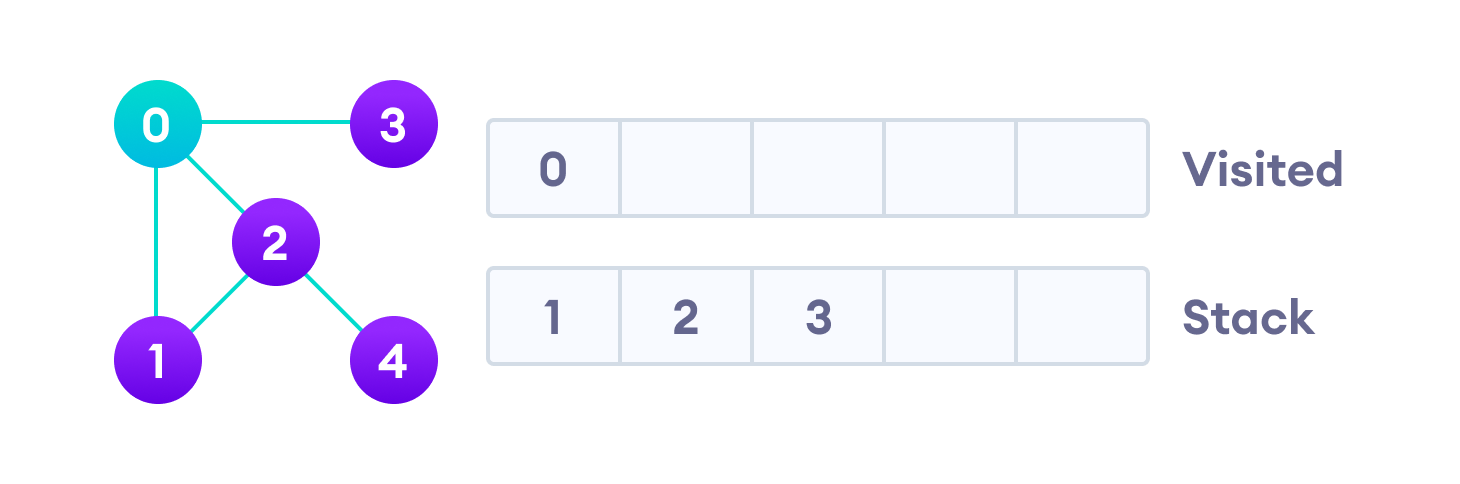
Алгоритм поиска в глубину работает следующим образом:

1. Начните с того, что поместите любую вершину графа на вершину стека.
2. Возьмите верхний элемент стека и добавьте его в список “Пройденных”.
3. Создайте список смежных вершин для этой вершины. Добавьте те вершины, которых нет в списке “Пройденных”, в верх стека.
4. Необходимо повторять шаги 2 и 3, пока стек не станет пустым.

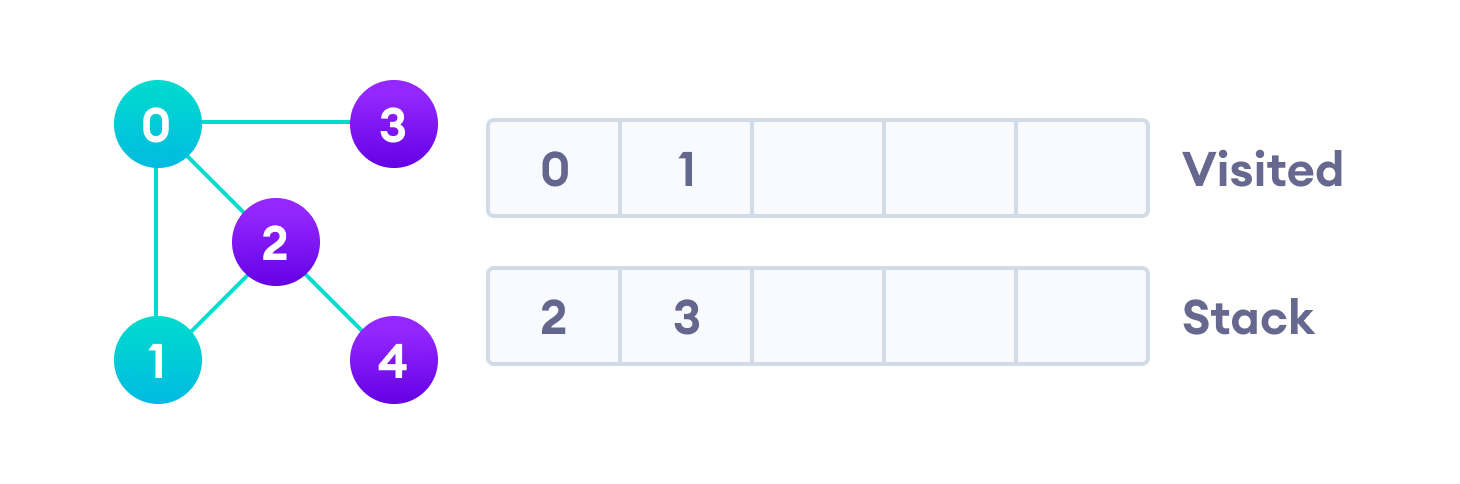
**Пример реализации поиска в глубину**

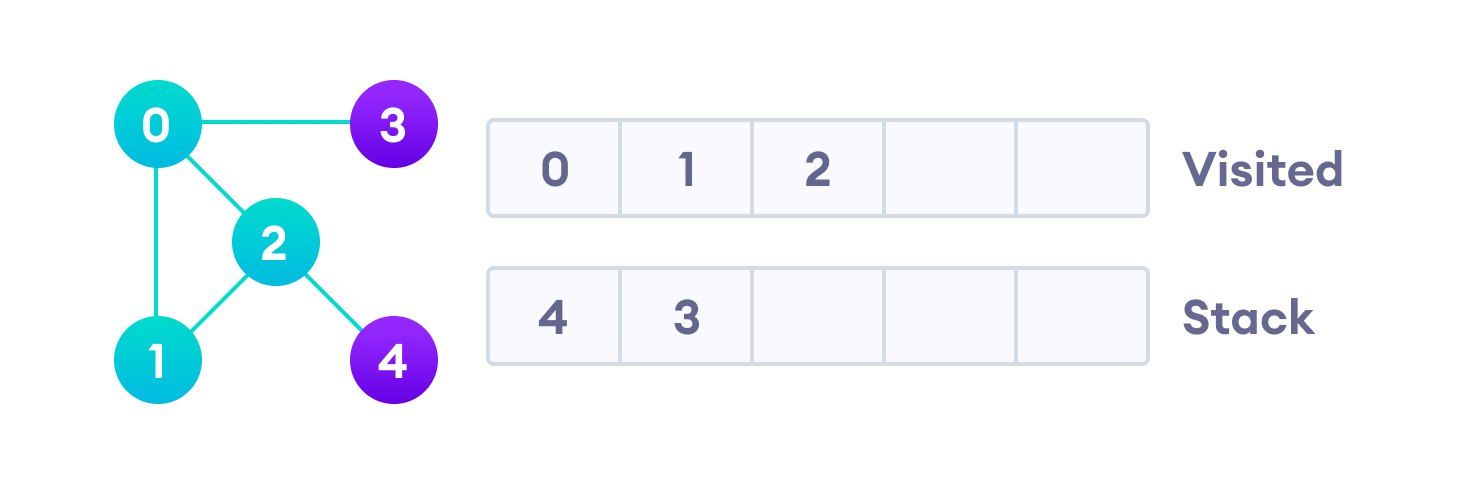


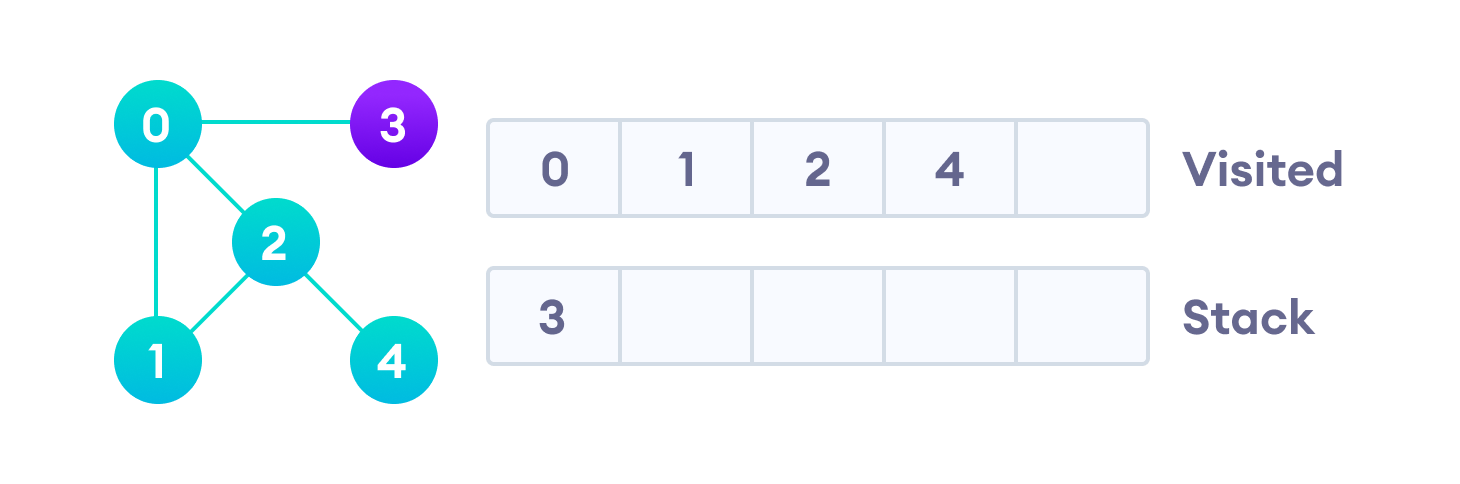
Начнем мы с вершины “0”. В первую очередь алгоритм поиска в глубину поместит ее саму в список “Пройденные” (на изображении “Visited”), а ее смежные вершины — в стек.



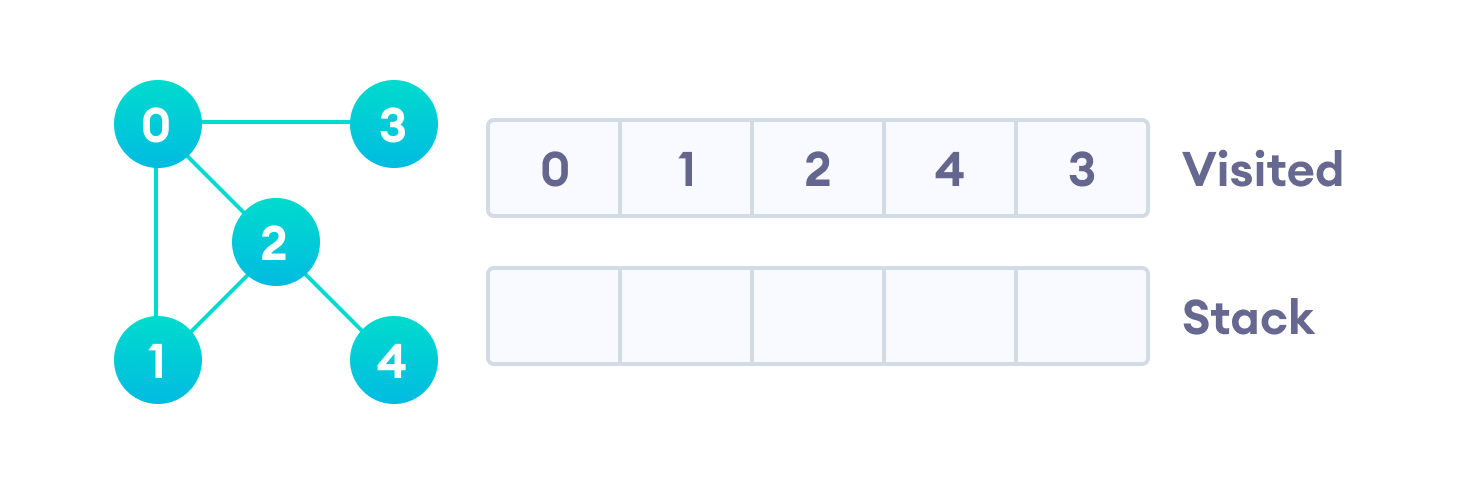
Затем мы берем следующий элемент сверху стека, т.е. к вершину “1”, и переходим к ее соседним вершинам. Поскольку вершина “0” уже пройдена, следующая вершина “2”.

 Вершина “2” смежна непройденной вершине “4”, следовательно мы добавляем ее наверх стека и проходим ее.





После того, как мы пройдем последний элемент (вершину “3”), в стеке не останется непройденных смежных вершин, и таким образом мы завершили обход графа в глубину.



**Сложность алгоритма поиска в глубину**

Временная сложность алгоритма поиска в глубину представлена ​​в виде O(V + E), где V — количество вершин, а E — количество ребер.

**Структурное программирование**

является методом проектирования программного обеспечения, который предполагает разбиение программы на относительно небольшие блоки, называемые модулями. Эти модули должны быть логически связаны между собой и иметь четко определенные интерфейсы. Применение структурного программирования при разработке программного обеспечения, использующего алгоритмы глубокого поиска пути, может помочь улучшить его качество и сделать его более эффективным. Например, модуль, который реализует алгоритм A\*, может быть разделен на несколько подмодулей, каждый из которых отвечает за определенную часть алгоритма. Это может сделать код более читаемым и упростить его поддержку. Таким образом, алгоритмы глубокого поиска пути и структурное программирование могут быть очень полезными при разработке программного обеспечения для задач компьютерного зрения. Они могут помочь улучшить качество и эффективность программного обеспечения, а также упростить его поддержку и разработку.

**Особенности поиска в глубину**

Алгоритмы глубокого поиска пути, такие как поиск в глубину, имеют несколько особенностей:

1. Они исследуют графы путей до тех пор, пока не найдут решение или не пройдут все возможные пути. Это может занять много времени и ресурсов, особенно если граф очень большой.
2. Они могут столкнуться с проблемой зацикливания, когда они заходят в бесконечный цикл и не могут продолжать поиск. Чтобы избежать этой проблемы, можно использовать различные стратегии, такие как отслеживание посещенных вершин или ограничение глубины поиска.
3. Они могут не находить оптимальное решение, если они исследуют только часть графа путей. Для получения оптимального решения может потребоваться использование других алгоритмов, таких как алгоритм Дейкстры или алгоритм A\*.
4. Они могут быть подвержены проблемам с памятью, особенно если граф очень большой. Для решения этой проблемы можно использовать различные стратегии, такие как использование итеративного углубления или ограничение размера стека.

**Алгоритм А\***

Алгоритм A\* является улучшенной версией алгоритма поиска в глубину, который использует эвристическую функцию для оценки расстояния до цели. Он основан на идее поиска пути с минимальной стоимостью, где стоимость определяется суммой стоимости пути от начальной вершины до текущей вершины и эвристической оценки расстояния от текущей вершины до цели.

Алгоритм A\* работает следующим образом:

1. Создать список открытых вершин и добавить в него начальную вершину. Создать список закрытых вершин.

2. Пока список открытых вершин не пуст:

a. Выбрать вершину с наименьшей стоимостью из списка открытых вершин.

b. Если выбранная вершина является целевой, то построить путь от начальной вершины до целевой и вернуть его.

c. Переместить выбранную вершину из списка открытых вершин в список закрытых вершин.

d. Для каждой соседней вершины, которая еще не была посещена:

i. Рассчитать стоимость пути от начальной вершины до соседней вершины.

ii. Рассчитать эвристическую оценку расстояния от соседней вершины до цели.

iii. Добавить соседнюю вершину в список открытых вершин, если ее там нет, или обновить ее стоимость, если она уже есть в списке открытых вершин.

3. Если список открытых вершин пуст и целевая вершина не найдена, то путь не существует.

Алгоритм A\* является эффективным и оптимальным алгоритмом поиска пути, который может использоваться для различных задач, таких как поиск кратчайшего пути на картах или в играх. Однако, для его работы необходимо иметь эвристическую функцию, которая может быть сложной в некоторых случаях.

# **Реализация алгоритма**

Алгоритм поиска в глубину (DFS - Depth-First Search) используется для обхода графа или дерева. Он начинает с одной вершины и идет в глубину, пока не достигнет конечной вершины или не пройдет все вершины. Алгоритм:

1. Поместить начальную вершину в стек.
2. Если стек пуст, то завершить поиск.
3. Извлечь вершину из стека.
4. Если эта вершина является конечной, то завершить поиск.
5. Если эта вершина не была посещена, то пометить ее как посещенную и добавить все ее соседние вершины в стек.
6. Повторить шаги 2-5.

# **Пример работы**

1. Используем программу «gen\_lab\_origin» для создания файла «maze-for-u» и получаем лабиринт

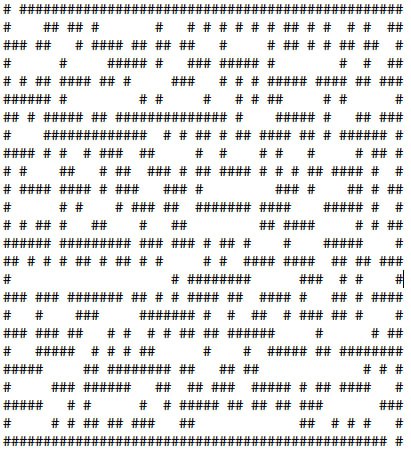


Рисунок 1. Результат работы программы «gen\_lab\_origin»

1. Ставим ключ «\*» в произвольное место.
2. Используем программу под названием «laba 3» после ее запуска появляется текстовый файл «maze-for-me-done»
3. Открываем текстовый файл и получаем результат:

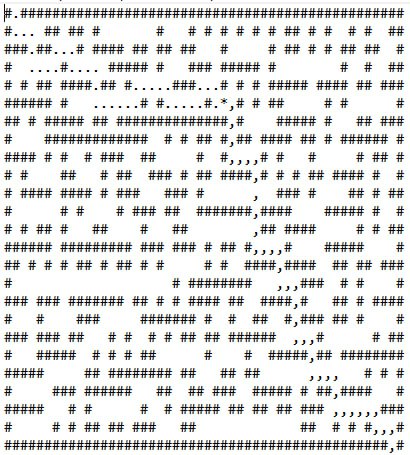


Рисунок 2. Результат работы программы «laba 3»

# **Заключение**

Алгоритмы глубокого поиска пути являются одними из наиболее распространенных алгоритмов в области компьютерного зрения и машинного обучения. Они могут быть использованы для решения различных задач, таких как поиск кратчайшего пути между двумя вершинами, поиск пути с минимальной стоимостью и т.д. Структурное программирование, в свою очередь, является методом проектирования программного обеспечения, который может помочь упростить код и сделать его более понятным и легко поддерживаемым. Применение структурного программирования при разработке программного обеспечения, использующего алгоритмы глубокого поиска пути, может помочь улучшить его качество и сделать его более эффективным.

# **Список литературы**

1. Поиск в глубину[Электронный ресурс]. URL: <https://ru.wikipedia.org/wiki/Поиск_в_глубину> (Дата обращения: 13.05.2023).
2. Алгоритм A\*[Электронный ресурс]. URL: [https://ru.wikipedia.org/wiki/A\*](https://ru.wikipedia.org/wiki/A*) (Дата обращения: 13.05.2023).
3. Обход графа[Электронный ресурс]. URL: <https://habr.com/ru/articles/504374/> (Дата обращения: 13.05.2023).
4. Алгоритм поиска в глубину [Электронный ресурс]. URL:  
   <https://habr.com/ru/companies/otus/articles/660725/> (Дата обращения: 13.05.2023).
5. Метод поиска пути в лабиринте при наличии помех [Электронный ресурс]. URL: (<https://cyberleninka.ru/>) (Дата обращения: 13.05.2023).
6. Depth-First Search - поиск в глубину [Электронный ресурс]. URL: (<https://blog.skillfactory.ru/glossary/dfs/>) (Дата обращения: 13.05.2023).
7. Основные структуры данных [Электронный ресурс]. URL: <https://habr.com/ru/articles/422259/> (Дата обращения: 13.05.2023).
8. Алгоритм поиска пути и А\* [Электронный ресурс]. URL: <https://habr.com/ru/articles/444828/> (Дата обращения: 13.05.2023).

# **Приложение 1**

## Листинг программы

#функция для преобразования лабиринта

def readmaze(filename):

with open(filename)as f:

lines=f.read().splitlines()

maze=[list(line)for line in lines]

return maze

#поиск точек начала и конца

def points\_start\_end(maze):

start = None

end = None

for y in range(len(maze[0])):

if maze[0][y]==" ":

start=(0,y)

if maze[len(maze)-1][y]==" ":

end=((len(maze)-1),y)

return start,end

#Поиск ключа

def key\_point(maze):

key= None

for i in range(len(maze)):

for j in range(len(maze[0])):

if maze[i][j]== "\*":

key = (i,j)

break

return key

#получение соседних точек

def get\_neighbor(maze,point):

row, col=point

neighbors=[]

if row > 0 and maze[row-1][col] !="#":

neighbors.append((row-1,col))

if row < len(maze)-1 and maze[row+1][col] != "#":

neighbors.append((row+1,col))

if col > 0 and maze[row][col-1] != "#":

neighbors.append((row,col-1))

if col < len(maze[0])-1 and maze[row][col+1] !="#":

neighbors.append((row,col+1))

return neighbors

#Поиск пути с помощью DFS

def DFS(maze,start,end):

#создаем стек и добавляем наяальную точку в него

stack=[start]

#создаем словарь для оьслеживания посещенных точек

visited={start:None}

while stack:

#извлекаем вершину из стека

current=stack.pop()

#если достигли конечной точки,то возвращаем путь

if current==end:

path=[]

while current is not None:

path.append(current)

current=visited[current]

return path[::-1]

#добавляем в стек все соседние точки, которые еще не были посещены

for neighbor in get\_neighbor(maze,current):

if neighbor not in visited:

stack.append(neighbor)

visited[neighbor]=current

#если не нашли путь

return None

#объявление лабиранта с пометкой пути

def new\_maze(maze,path,simvol):

for coordinata in path:

x,y=coordinata

maze[x][y]=simvol

return maze

#запись лабиринта в отдельный файл

def new\_file(maze,filename):

with open(filename,"w")as f:

for row in maze:

for elem in row:

f.write(str(elem))

f.write("\n")

maze=readmaze("maze-for-u.txt")

start,end=points\_start\_end(maze)

key=key\_point(maze)

path1=DFS(maze,start,key)

path2=DFS(maze,key,end)

maze=new\_maze(maze,path1,".")

maze=new\_maze(maze,path2,",")

x,y=key

maze[x][y]="\*"

new\_file(maze,"maze-for-me-done.txt")