Минобрнауки России

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

(ННГАСУ)

Кафедра информационных систем и технологий

Курсовая работа

по дисциплине: «Язык программирования Python»

Алгоритмы поиска пути и структурное программирование

Выполнил студент 1 курса гр. ИС-33

Соболев Д.С.

Проверил Преподаватель

Морозов Н.С.

Нижний Новгород

2023

Оглавление

[**Введение** 3](#_Toc137176983)

[**Цель работы:** 3](#_Toc137176984)

[**Задачи:** 3](#_Toc137176985)

[**1.** **Теоретическая часть** 3](#_Toc137176986)

[1.1 Алгоритмы поиска пути 4](#_Toc137176987)

[1.1.1 Поиск в глубину (DFS) 4](#_Toc137176988)

[1.1.2 Алгоритм А\* 6](#_Toc137176989)

[1.2 Структурное программирование 7](#_Toc137176990)

[**2.** **Реализация алгоритма** 8](#_Toc137176991)

[**Результат работы алгоритма** 9](#_Toc137176992)

[**Заключение** 10](#_Toc137176993)

[**Список литературы** 11](#_Toc137176994)

[**Приложение 1** 12](#_Toc137176995)

[Листинг программы 12](#_Toc137176996)

# **Введение**

Алгоритмы поиска пути являются важной частью программирования. С помощью них находят оптимальный маршрут между двумя точками на графе или в сети. Например, такие алгоритмы используются в навигационных системах, чтобы определить кратчайший путь от точки А до точки Б, в сетевых технологиях, чтобы определить наилучший маршрут передачи пакетов, в играх, чтобы определить путь противника или игрока к цели, и многих других приложениях. Они помогают оптимизировать ресурсы и сократить время на выполнение задач. Структурное программирование – это методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. В данной работе будут рассмотрены и использованы одни из основных алгоритмов поиска пути – поиск в глубину (Depth First Search) и А\* с применением элементов структурного программирования.

# **Цель работы:**

Изучить и написать программу на языке программирования Python для нахождения пути в лабиринте, в которой будут использованы алгоритмы поиска пути и структурное программирование

# **Задачи:**

1. Изучить и использовать алгоритмы поиска пути на примере DFS и А\*
2. Ознакомиться с понятием структурного программирования
3. Написать программу для нахождения пути в лабиринте на языке программирования Python
4. Сохранить файл с построенным маршрутом в лабиринте
5. **Теоретическая часть**
   1. Алгоритмы поиска пути
      1. Поиск в глубину (DFS)

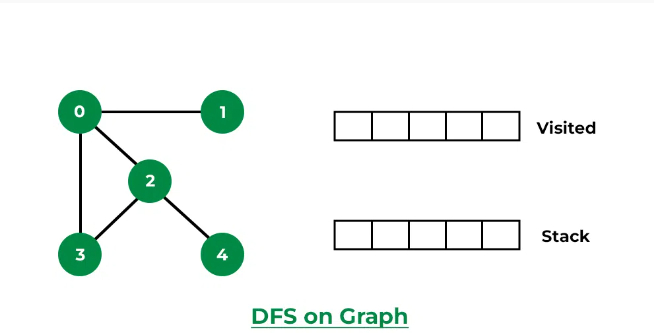
**Алгоритм поиска (или обхода) в глубину** (англ. depth-first search, DFS) позволяет построить обход ориентированного или неориентированного графа, при котором посещаются все вершины, доступные из начальной вершины.

Алгоритм DFS начинает обходить граф с любой случайной вершины и продвигается путем перехода в следующую вершину, которая не посещалась ранее. Как только не остается неисследованных вершин у текущей, алгоритм возвращает управление, чтобы продолжить из следующей вершины. DFS может быть реализован с использованием рекурсии или стека. Во время обхода графа DFS помечает каждую вершину, которую он посещает, чтобы избежать зацикливания при обходе графа.

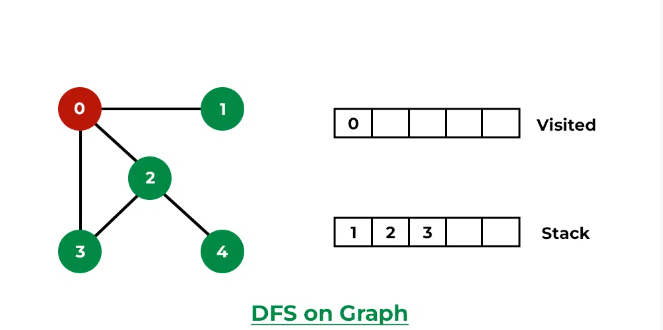
То есть, основной идеей DFS является полное изучение одной ветки, после чего уже можно переходить другим, при условии, что пути делятся на несколько путей, которые в свою очередь выходят из начала. [1]

Реализация алгоритма в глубину:

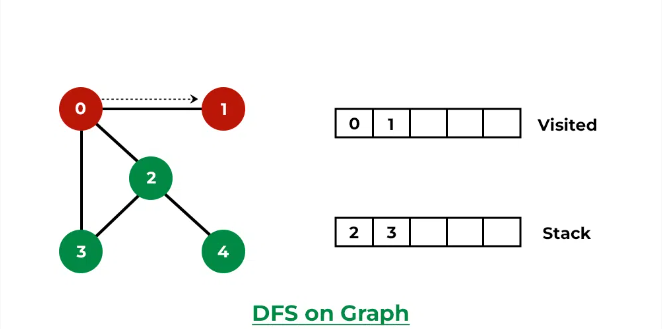
Шаг 1: Создаем массивы stack и visited, которые изначально пусты.



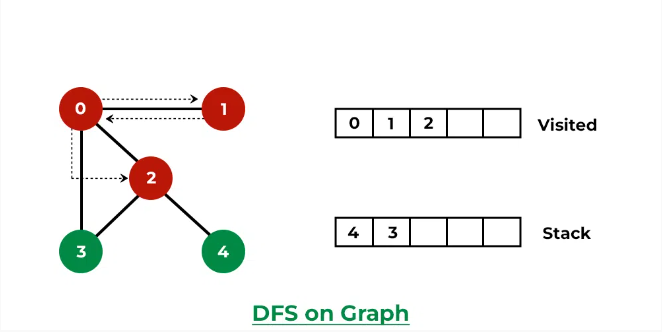
Шаг 2: Посетим 0 и поместим его соседние узлы, которые еще не посещены, в стек.



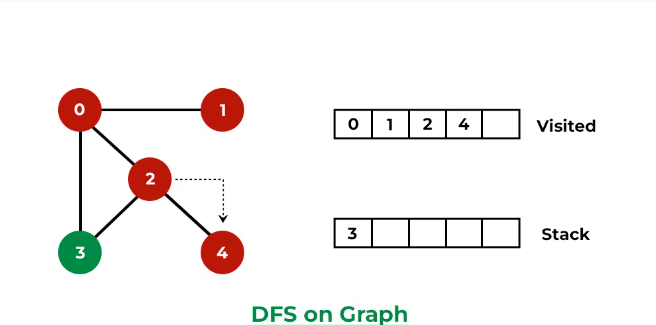
Шаг 3: Теперь узел 1 находится в верхней части стека, поэтому посетим узел 1 и извлечем его из стека и поместим все его соседние узлы, которые не посещались, в стек.



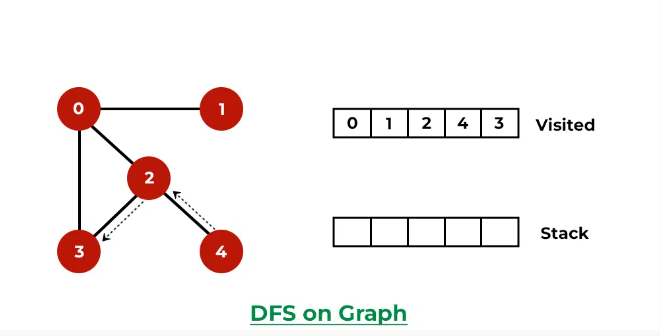
Шаг 4: Теперь узел 2 находится в верхней части стека, поэтому посетим узел 2 и извлечем его из стека и поместим все его соседние узлы, которые не посещались (т.е. 3, 4) в стек.



Шаг 5: Теперь узел 4 находится в верхней части стека, поэтому посетим узел 4 и извлечем его из стека и поместим все его соседние узлы, которые не посещались, в стек.



Шаг 6: Теперь узел 3 находится в верхней части стека, поэтому посетим узел 3 и извлечем его из стека и поместим все его соседние узлы, которые не посещались, в стек. [2]



* + 1. Алгоритм А\*

Алгоритм А\* - это эвристический поисковый алгоритм, который используется для нахождения кратчайшего пути между двумя точками на графе. Он работает на основе поиска в глубину и использует функцию оценки стоимости для выбора наилучшего пути для расширения.

Алгоритм A\* был разработан для поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной). Он похож на алгоритм Дейкстры, но его подход гораздо более целенаправленный. Для целевого узла он сначала вычисляет оценку кратчайшего расстояния. Основываясь на этой оценке, он обеспечит быстрое вычисление. [3]

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)*). [4]

* 1. Структурное программирование

**Структурным программированием** называется разработка программ с помощью представления их в виде иерархической структуры блоков. Эта парадигма разработана в 70-х годах XX века Э. Дейкстрой и Н. Виртом.[5]

Условия структурного программирования включают в себя:

1. Любая программа представляет собой структуру, построенную из трёх типов базовых конструкций:
   * последовательное исполнение — однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
   * ветвление — однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;
   * цикл — многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).
2. Повторяющиеся фрагменты программы (либо не повторяющиеся, но представляющие собой логически целостные вычислительные блоки) могут оформляться в виде т. н. подпрограмм (процедур или функций). В этом случае в тексте основной программы, вместо помещённого в подпрограмму фрагмента, вставляется инструкция **вызова подпрограммы**. При выполнении такой инструкции выполняется вызванная подпрограмма, после чего исполнение программы продолжается с инструкции, следующей за командой вызова подпрограммы.
3. Разработка программы ведётся пошагово, методом «сверху вниз».

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. [5]

1. **Реализация алгоритма**

Данный программный код должен получать текстовый файл с лабиринтом, определить, где находиться начало, конец и ключ, найти путь с помощью DFS и А\* и записать в новый файл лабиринт с пройденным маршрутом.

Сначала импортируем модули, которые нам понадобятся:

- import sys для работы с различными функциями и переменными

- import heapq для реализации приоритетных очередей – структуры данных.

Применяемые функции:

- Функция def dfs(curr) используется для обхода графа в глубину, которая начинает поиск с вершины curr. Функция просматривает все вершины в глубину, начиная с данной вершины. Каждая вершина отмечается, как пройденная, чтобы избежать бесконечного цикла.

- Функция heuristic(curr, end) - это эвристическая функция, которая используется в алгоритмах поиска пути, например, A\*. Она оценивает оставшееся расстояние от текущей точки до конечной точки (цели) и помогает выбрать оптимальный путь для достижения цели.

- Функция def readMaze() используется для чтения лабиринта из файла, координаты стен и проходов в лабиринте, указывая, где находится начальный и конечный пункты, а также другие параметры.

- Функция def writeMaze() выводит лабиринт с путями в новый текстовый файл.

Основная часть программы:

1) Открываем файл maze-for-u.txt, в котором сгенерирован наш лабиринт, и считываем его в матрицу.

2) Идет работа над поиском координаты начальной точки и координаты ключа.

3) От начальной точки до ключа, программа выполняет поиск в глубину.

4) Если путь от начальной точки до ключа найден, то он заполняется точками, если нет, то возвращаем False.

5) Идет работа над поиском координаты конечной точки.

6) От ключа до конечной точки, программа выполняет поиск по алгоритму А\*.

7) Если путь от ключа до конечной точки найден, то он заполняется запятыми, если нет, то возвращаем пустой список.

8) Сохраняем получившийся лабиринт в maze-for-me-done.txt

# **Результат работы алгоритма**

При запуске, программа читает лабиринт, находит путь от начальной точки к ключу, используя алгоритмы поиска DFS находит путь от начальной точки до ключа, потом ищет путь от ключа до конечной точки и используя алгоритм поиска A\* доходит до конца. Программа заполняет пустые ячейки в лабиринте нужными символами и сохраняет заполненный лабиринт в новый текстовый файл ‘maze-for-me-done.txt’.

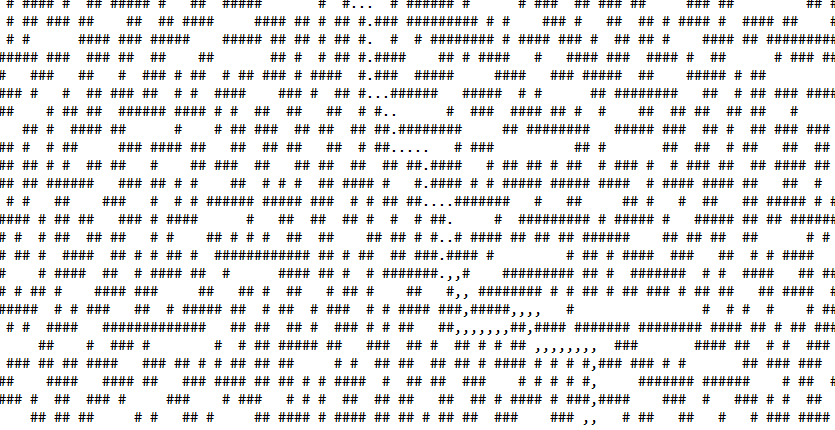
****

Рис. 1. Результат работы программы

# **Заключение**

В данной работы были рассмотрены основные алгоритмы поиска пути, такие как поиск в глубину (DFS) и А\*, с помощью которых были найдены варианты продвижения в лабиринте. Были рассмотрены и изучены основы алгоритмов поиска в глубину (DFS) и алгоритма А\*, а также было рассмотрено структурное программирование.

Была написана программа на языке программирования Python, где использовались алгоритм поиска в глубину (DFS) и алгоритма А\*. С их помощью был найден путь прохождения лабиринта от начала до ключа и от ключа до конца. Результат работы программы был сохранен в текстовый файл

# **Список литературы**

* 1. Алгоритм поиска в глубину / [Электронный ресурс]. URL: <https://foxford.ru/wiki/informatika/algoritm-poiska-v-glubinu> (Дата обращения 08.06.2023)
  2. Поиск по глубине или DFS для графика / [Электронный ресурс]. URL: <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/> (Дата обращения 08.06.2023)
  3. Лорьер Ж.-Л. Системы искусственного интеллекта: Пер. с франц.— М.: Мир, 1991. - 568 с. Глава 5, тема 5.2. Алгоритм А\*
  4. Алгоритм А\* и его реализация на Python / [Электронный ресурс]. URL: https://pythonist.ru/algoritm-a-star-i-ego-realizacziya-na-python/ (Дата обращения 08.06.2023)
  5. Парадигма структурного программирования / [Электронный ресурс]. URL: <https://foxford.ru/wiki/informatika/paradigma-strukturnogo-programmirovaniya> (Дата обращения 08.06.2023)
  6. The Python Standart Library / [Электронный ресурс]. URL: <https://docs.python.org/3/library/index.html> (Дата обращения 08.06.2023)
  7. Обход графа: поиск в глубину / [Электронный ресурс]. URL: <https://habr.com/ru/articles/504374/> (Дата обращения 08.06.2023)

# **Приложение 1**

## Листинг программы

import sys

import heapq

sys.setrecursionlimit(10000)

# Функция с алгоритмом поиска пути в глубину в лабиринте от точки Start до точки End

def depth\_first\_search(maze, height, width, start, end):

def dfs(curr):

nonlocal visited, path

if curr == end: # если достигли конечной точки, возвращаем путь

path.append(curr)

return True

visited.add(curr) # добавляем текущую вершину в посещенные

# перебираем соседние вершины

for dx, dy in ((-1, 0), (1, 0), (0, -1), (0, 1)):

x, y = curr[0] + dx, curr[1] + dy

if 0 <= x < height and 0 <= y < width and maze[x][y] == 1 and (x, y) not in visited:

# если соседняя вершина проходима и еще не была посещена

if dfs((x, y)): # рекурсивно вызываем функцию для этой вершины

path.append(curr)

return True

return False # если не нашли путь, возвращаем False

visited = set() # множество посещенных вершин

path = [] # список вершин на пути

dfs(start)

return path[::-1] # возвращаем путь в обратном порядке

# Функция с алгоритмом A\* поиска пути в лабиринте от точки Start до точки End

def a\_star(maze, height, width, start, end):

def heuristic(curr, end):

# эвристическая функция - оценка расстояния до целевой точки

return abs(curr[0] - end[0]) + abs(curr[1] - end[1])

pq = [] # очередь с приоритетом

parent = {} # словарь для хранения предков вершин

g\_score = {start: 0} # словарь для хранения стоимости пути до вершины

f\_score = {start: heuristic(start, end)} # словарь для хранения эвристической оценки стоимости пути

heapq.heappush(pq, (f\_score[start], start)) # добавляем начальную вершину в очередь

while pq:

curr\_f, curr = heapq.heappop(pq)

if curr == end: # если достигли конечной точки, возвращаем путь

path = []

while curr in parent:

path.append(curr)

curr = parent[curr]

path.append(start)

return path[::-1]

# перебираем соседние вершины

for dx, dy in ((-1, 0), (1, 0), (0, -1), (0, 1)):

x, y = curr[0] + dx, curr[1] + dy

if 0 <= x < height and 0 <= y < width and maze[x][y] == 1:

# если соседняя вершина проходима

next\_node = (x, y)

new\_g\_score = g\_score[curr] + 1 # стоимость пути до соседней вершины

if next\_node not in g\_score or new\_g\_score < g\_score[next\_node]:

# если соседняя вершина еще не обрабатывалась или найден новый более короткий путь до нее

parent[next\_node] = curr

g\_score[next\_node] = new\_g\_score

f\_score[next\_node] = new\_g\_score + heuristic(next\_node, end)

heapq.heappush(pq, (f\_score[next\_node], next\_node))

return [] # если не нашли путь, возвращаем пустой список

def readMaze():

maze = []

# Открываем файл с лабиринтом и считываем его в матрицу

with open("maze-for-u.txt", "r") as f:

file = f.readlines()

height = len(file)

width = len(file[0]) - 1

for line in file:

tmp\_line = [0 if sym == "#" else 1 for sym in line[:-1]]

maze.append(tmp\_line[:])

return maze, height, width

def writeMaze():

# Выводим лабиринт с путями в текстовый файл

with open("maze-for-me-done.txt", "w") as f:

for x in range(height):

for y in range(width):

if (x, y) == key\_pos:

f.write("\*")

elif (x, y) in path\_to\_key:

f.write(".")

elif (x, y) in path\_to\_exit:

f.write(",")

elif maze[x][y] == 0:

f.write("#")

else:

f.write(" ")

f.write("\n")

maze, height, width = readMaze()

# Ввод исходных данных

print(f"Координаты аватара через пробел: ", end="")

avatar\_pos = tuple([int(\_) for \_ in input().split()])

print(f"\nКоординаты ключа через пробел: ", end="")

key\_pos = tuple([int(\_) for \_ in input().split()])

print(f"\nКоординаты выхода через пробел: ", end="")

exit\_pos = tuple([int(\_) for \_ in input().split()])

print("Координаты аватара через пробел: 0 1")

print("Координаты ключа через пробел: 466 592")

print("Координаты выхода через пробел: 599 798")

avatar\_pos = 0, 1

key\_pos = 466, 592

exit\_pos = 599, 798

# Вычисляем пути от входа до ключа и от ключа до выхода

path\_to\_key = depth\_first\_search(maze, height, width, avatar\_pos, key\_pos)

path\_to\_exit = a\_star(maze, height, width, key\_pos, exit\_pos)

writeMaze()