Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Нижегородский государственный архитектурно-строительный университет»

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,

ФИЭСИС, группы ИС-33

Цветков А.С.

Проверил: преподаватель,

Морозов Н.С.

Нижний Новгород, 2023 г.

# Оглавление

[Введение 3](#_Toc136671914)

[Задачи 3](#_Toc136671915)

[Теоретическая часть 4](#_Toc136671916)

[Для чего нужен алгоритм DFS 4](#_Toc136671917)

[Реализация алгоритма 7](#_Toc136671918)

[Пример работы 8](#_Toc136671919)

[Заключение 9](#_Toc136671920)

[Список литературы 10](#_Toc136671921)

[Приложение 11](#_Toc136671922)

[Листинг программы 11](#_Toc136671923)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании, а графы являются существенным элементом математических моделей в самых разнообразных областях науки и практики. Они помогают наглядно представить взаимоотношения между объектами или событиями в сложных системах.

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

## Задачи

* Изучить алгоритмы построения маршрута в графе;
* Выделить особенности, необходимые в конкретной задаче поиска маршрута;
* Подготовить лабиринт и исходные координаты точек;
* Написать алгоритмы с заданными параметрами;
* Сохранить результаты обходов лабиринта и маршруты в файл.

# Теоретическая часть

## Для чего нужен алгоритм DFS

Классическая реализация DFS — рекурсивная. Рекурсией называется процесс, когда какая-то функция вызывает себя же, но с другими аргументами. В результате одна и та же функция одновременно остается запущенной несколько раз, пока в одной из итераций не дойдет до финального решения. Тогда она вернет результат и закроется, а по каскаду закроются и все функции, приведшие к ее вызову.

С помощью рекурсии решают задачи, связанные с “глубоким” прохождением по данным. Очевидный пример — расчет чисел Фибоначчи, когда на каждом следующем шаге функция складывает два предыдущих числа. Задача DFS тоже классически решается с помощью рекурсивного алгоритма.

**Первый шаг.** Когда алгоритм начинает работу, все вершины считаются “белыми”, непосещенными. DFS начинает путь в заранее заданной вершине v и должен найти от нее путь до другой заданной вершины или же полностью составить карту графа.

Первое, что делает DFS, — красит вершину, в которой находится, в серый цвет. Это показывает, что алгоритм в ней уже был. Затем DFS проверяет соседей — вершины, которые соединены с той, где он находится.

**Переход.** Если какая-то из смежных вершин белая, алгоритм переходит на нее и повторяет те же действия: красит в серый, ищет соседей. Это происходит не циклически, а рекурсивно: если представить DFS как функцию, то получится, что эта функция в ходе выполнения вызывает сама себя, но для другой вершины. Поэтому сначала алгоритм работает с одним выбранным соседом и, только если упирается в тупик, возвращается обратно и пробует пройти по другому пути.

Выбор соседа происходит случайно или по заранее заданным критериям — например, это может быть самая левая или самая правая вершина. Выше мы упоминали “правило левой руки”: оно по сути является таким критерием.

Если неисследованных соседей у вершины не осталось, она красится в черный цвет как полностью посещенная.

**Завершение обхода.** Алгоритм завершается, если достигает нужной точки. В таком случае все вызванные “экземпляры” функции поочередно завершаются: от последнего до первого вызванного. Если задача — полностью перебрать граф, то критерий для завершения другой: все вершины должны стать черными [Рис. 1.].

Для чего применяется:

* Для поиска любого маршрута в лабиринте. В отличие от алгоритма BFS, поиск в глубину ищет не самый короткий, а случайный путь. Правило прохождения лабиринта в реальной жизни “Идти с левой рукой на стене и всегда поворачивать влево” — пример DFS вне программирования.
* 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. Мы подробно поговорим об этом ниже.

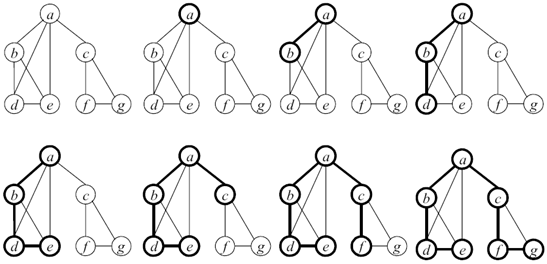


Рис. 1. Пример работы алгоритма DFS

# Реализация алгоритма

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.

Самая простая рекурсивная реализация — создать граф в виде связанного списка или другой структуры данных, а потом написать функцию для его прохождения. Как реализовать сам граф — зависит от языка программирования: обычно используются типы, позволяющие хранить множество значений. Каждый элемент такого “комплексного” типа является вершиной, а внутри вершины хранятся ссылки на другие элементы или их номера — так реализованы пути.

Сама же функция, условно названная DFS (*v*), довольно простая и действует по следующей логике.

1. На вход поступает белая вершина *v*.
2. Вершина *v*окрашивается в серый.
3. Ищется вершина *w,* смежная с *v* и белая.
4. Изнутри DFS (*v*) рекурсивно вызывается DFS (*w*).
5. Когда функция завершается, вершина *v* окрашивается в черный.

“Окрашивание” можно реализовать с помощью какой-либо переменной внутри вершины: например, значение 0 — белая, 1 — серая, и так далее.

# 

# Пример работы

После запуска программа анализирует лабиринт и строит путь точками от входа до ключа и запятыми от ключа до выхода, используя алгоритмы DFS и A\*. Сохраняет полученный лабиринт в отдельный файл ‘result.txt’ [Рис. 2.].

Точками обозначается путь алгоритма DFS, запятыми A\*.

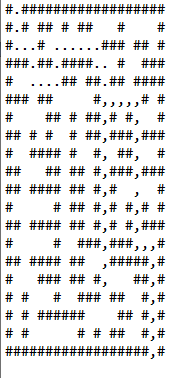


Рис. 2. Результат программы

# Заключение

В ходе работы изучили два алгоритма поиск в глубину DFS и алгоритм A\*. Они являются важными инструментами и помогают найти оптимальные или приближенные пути между различными точками в графах и сетях.

Параллельно с тем, как изучалась теория о заданных алгоритмах, была разработана программа, которая на основе изученной теории находила путь в лабиринте.

В результате было разработано консольное приложение для поиска выхода из лабиринта через ключ. Маршрут в виде схематического рисунка был сохранен в отдельный файл.

# Список литературы

1. Скиена С. Стивен Алгоритмы. Руководство по разработке. 3-е изд / Скиена С. Стивен. – СПб.: Springer, 2022. – 250 - 257, 848 с. – Текст:  
   непосредственный.
2. Алексеев В.Е. Таланов В.А.Графы Модели вычислений Структуры данных. / Издательство Нижегородского госуниверситета. – Нижний Новгород: 2004 – 4 c.– Текст: непосредственный.
3. OpenAIChatGPT [электронный ресурс] - <https://openai.com/blog/chatgpt> (Дата обращения: 11.05.2023)
4. Skillfactory [электронный ресурс] - <https://blog.skillfactory.ru>

(Дата обращения: 11.05.2023)

1. Habr [электронный ресурс] - <https://habr.com/ru/all/>

(Дата обращения: 11.05.2023)

1. Bestprog [электронный ресурс] - <https://brestprog.by/>

(Дата обращения: 11.05.2023)

1. EVILEG [электронный ресурс] - <https://evileg.com/ru/>

(Дата обращения: 11.05.2023)

# Приложение

## Листинг программы

from queue import PriorityQueue  
from math import sqrt  
import random  
  
  
def read\_maze(filename):  
 with open(filename) as f:  
 maze = [[char for char in line.strip()] for line in f]  
 return maze  
  
  
#В случайном месте создает ключ  
  
height = len(read\_maze("maze-for-u.txt"))  
width = len(read\_maze("maze-for-u.txt")[0])  
passages = []  
for i in range(height):  
 for j in range(width):  
 if read\_maze("maze-for-u.txt")[i][j] == " ":  
 passages.append((i, j))  
random\_key = random.choice(passages)  
def get\_neighbors(maze, cell: tuple[int, int]):  
  
 row, col = cell  
 neighbors = [(row - 1, col), (row + 1, col), (row, col - 1), (row, col + 1)]  
 valid\_neighbors = []  
 for neighbor in neighbors:  
 row, col = neighbor  
 if 0 <= row < len(maze) and 0 <= col < len(maze[0]) and maze[row][col] != "#":  
 valid\_neighbors.append(neighbor)  
 return valid\_neighbors  
  
  
# Поиск в глубину  
def find\_path(maze):  
 start = (0, 1)  
 key = random\_key  
  
 stack = [(start, [start])]  
 visited = set()  
 while stack:  
 current, path = stack.pop()  
 if current == key:  
 return path  
 visited.add(current)  
 for neighbor in get\_neighbors(maze, current):  
 if neighbor not in visited:  
 stack.append((neighbor, path + [neighbor]))  
 return None  
  
  
  
  
# А\*  
def get\_heuristic(cell, end):  
 #Эвристическое расстояние от ячейки до конечной точки  
  
 return sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2)  
  
  
def find\_path\_a\_star(maze):  
 key = random\_key  
 end = (len(maze) - 1, len(maze[0]) - 2)  
 queue = PriorityQueue()  
 queue.put((0, key, [key]))  
 visited = set()  
 while not queue.empty():  
 p, current, path = queue.get()  
 if current == end:  
 return p, path  
 visited.add(current)  
 for neighbor in get\_neighbors(maze, current):  
 if neighbor not in visited:  
 new\_path = path + [neighbor]  
 priority = len(new\_path) + get\_heuristic(neighbor, end)  
 queue.put((priority, neighbor, new\_path))  
 return None  
  
  
def main():  
 filename = "maze-for-u.txt"  
 maze = read\_maze(filename)  
 #Создание текстового документа, который рисует путь точками от входа до ключа и запятыми от ключа до выхода  
  
 path1 = find\_path(maze)  
 path2 = find\_path\_a\_star(maze)  
 path22 = path2[1]  
  
 for place in path1:  
 maze[place[0]][place[1]] = "."  
  
 result1 = ""  
 for line in maze:  
 result1 += "".join(line) + "\n"  
  
 for place in path22:  
 maze[place[0]][place[1]] = ","  
  
 result2 = ""  
 for line in maze:  
 result2 += "".join(line) + "\n"  
  
 with open("result.txt", "w") as f:  
 f.write(result2)  
main()