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 Гордеев А.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 12](#_Toc5)

[Заключение 13](#_Toc6)

[Список литературы 14](#_Toc7)

[Приложение 1 15](#_Toc8)

[Листинг программы 15](#_Toc9)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании [1]. Алгоритмы обхода графа являются важной задачей программирования, потому что они применяются во многих областях, включая компьютерные сети, базы данных, машинное обучение, анализ социальных сетей и многое другое. Эти алгоритмы помогают узнать о связях между элементами графа, находить оптимальные пути и выявлять различные сценарии взаимодействия между объектами. Одним из таких является поиск в глубину [2].

Более подробная информация о важности алгоритмов обхода графа может быть найдена в следующих источниках:

1. Обходы графов - Алгоритмика - Algorithmica (https://algorithmica.org/ru/dfs)

2.Алгоритм поиска в глубину - Информатика - Фоксфорд (<https://foxford.ru/wiki/informatika/algoritm-poiska-v-glubinu>)

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

# Задачи

* Изучить алгоритмы построения маршрута в графе;
* Выделить особенности реализации алгоритмов поиск в глубину и А\*;
* Подготовить исходные данные: лабиринт, координаты точек для посещения при обходе;
* Реализация алгоритмов на языке программирования Python;
* Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

# 1.Теоретическая часть

**Алгоритм A\***

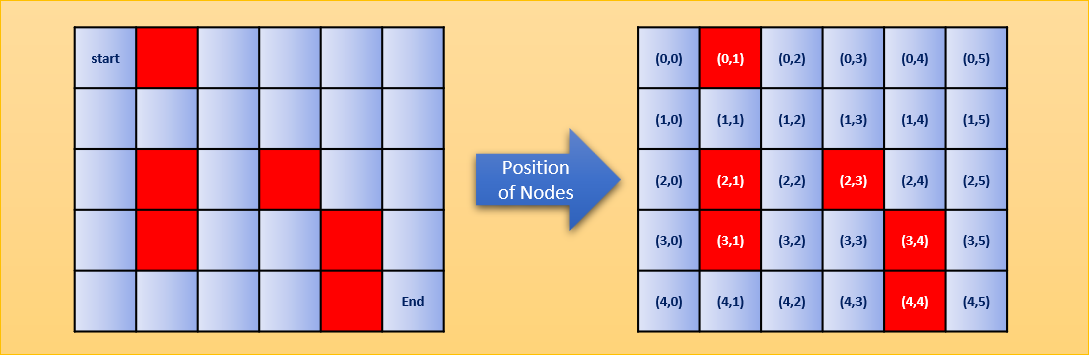
Алгоритм A\* является широко применяемым методом поиска кратчайшего пути в графе. Основной идеей алгоритма является оценка расстояния от текущей вершины до целевой вершины с помощью эвристической функции. Эта функция позволяет оценить стоимость пути от текущей вершины до цели через другие вершины графа. Алгоритм A\* учитывает как длину пройденного пути до текущей вершины, так и эвристическую оценку до цели.

Алгоритм A\* относится к группе эвристических методов поиска. По сравнению с жадным алгоритмом, он выбирает на каждом шаге не только наилучший вариант, но и учитывает путь до текущей вершины.

Алгоритм A\* активно применяется в различных областях, таких как машинное обучение, разработка игр и автономные транспортные средства. Например, при проектировании маршрутов для беспилотных машин, алгоритм A\* используется для нахождения оптимальных маршрутов с учетом всех ограничений и препятствий на пути.

Кроме того, алгоритм A\* может быть использован для решения задач оптимизации, например, для нахождения наименьшего количества операций, необходимых для перемещения объекта из одной точки в другую при игре в пятнашки или решении задачи коммивояжера.

В целом, алгоритм A\* является мощным и эффективным методом поиска кратчайшего пути в графе с широким спектром применения.



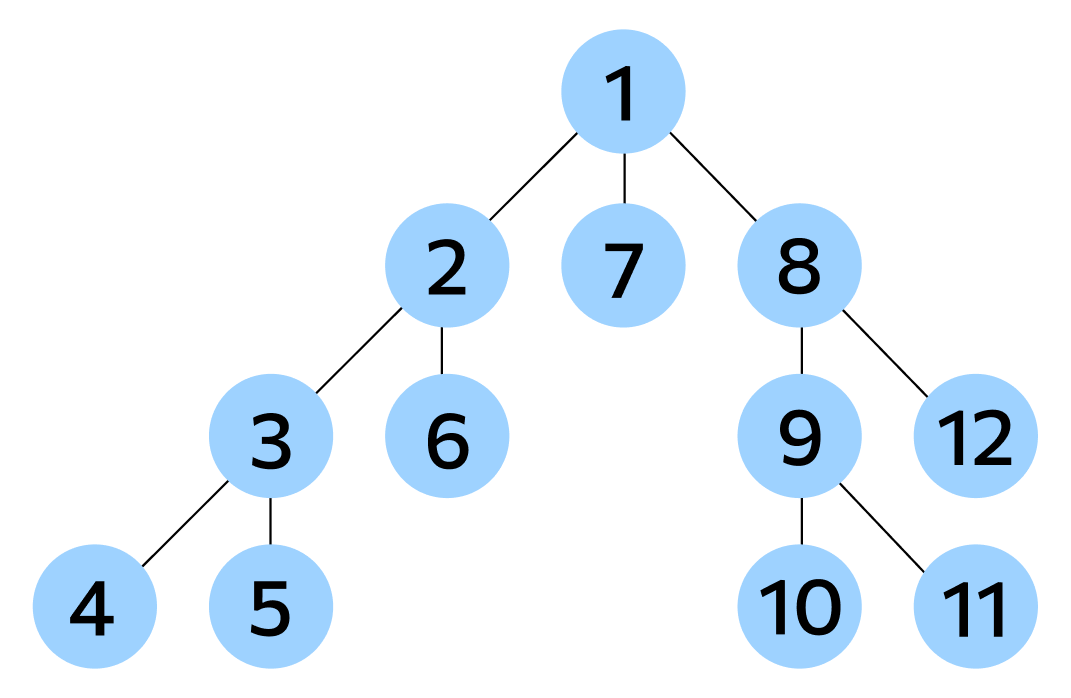
**Алгоритм поиска в глубину (DFS)**

Алгоритм поиска в глубину (DFS) является еще одним распространенным методом обхода графа и решения задачи о нахождении пути. В отличие от алгоритма A\*, DFS не гарантирует нахождения кратчайшего пути и может быть менее эффективным для поиска маршрутов в лабиринте. Принцип работы данного алгоритма следующий: текущую вершину мы отмечаем как посещенную, и для каждой соседней вершины заново реализуем алгоритм поиска. В этом и заключается рекурсивная составляющая поиска в глубину.

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.

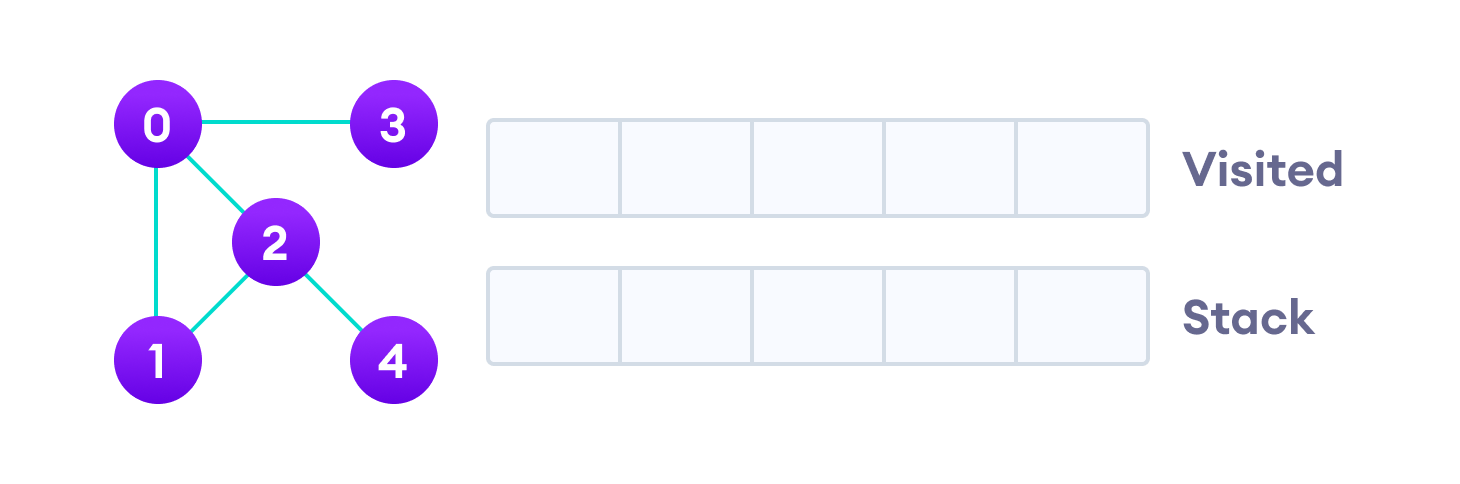
Для чего нужен алгоритм DFS:

* Для поиска любого маршрута в лабиринте. 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.
* 

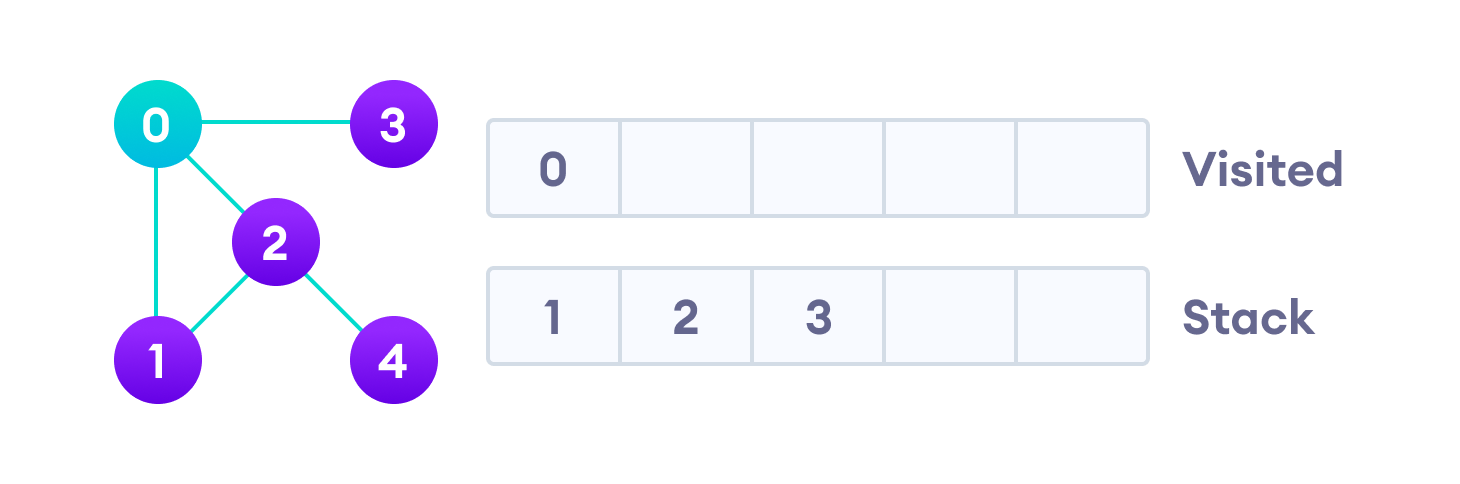
# 2. Реализация алгоритма

### **Пример реализации поиска в глубину**

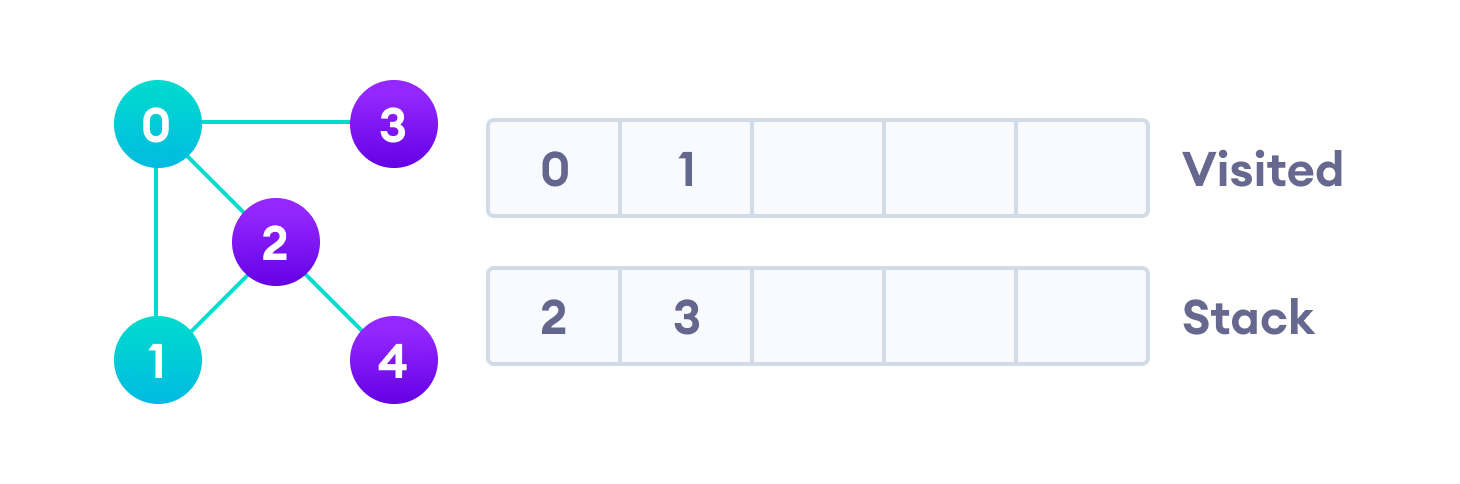
Предлагаю рассмотреть на примере, как работает алгоритм поиска в глубину. Мы будем использовать неориентированный граф с пятью вершинами.

Неориентированный граф с пятью вершинами

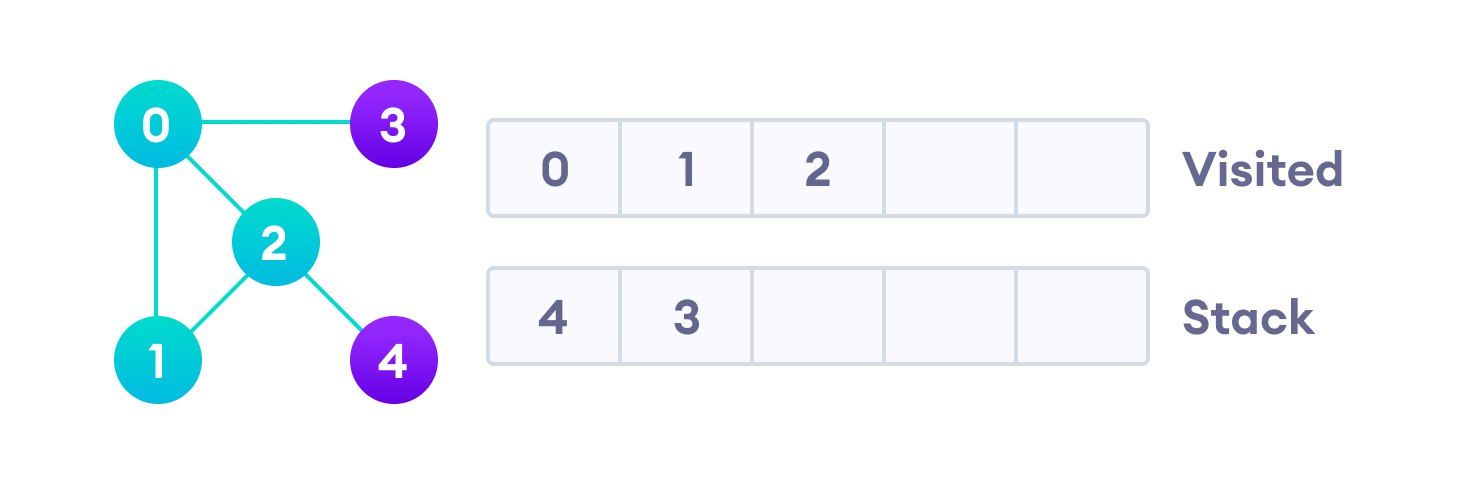
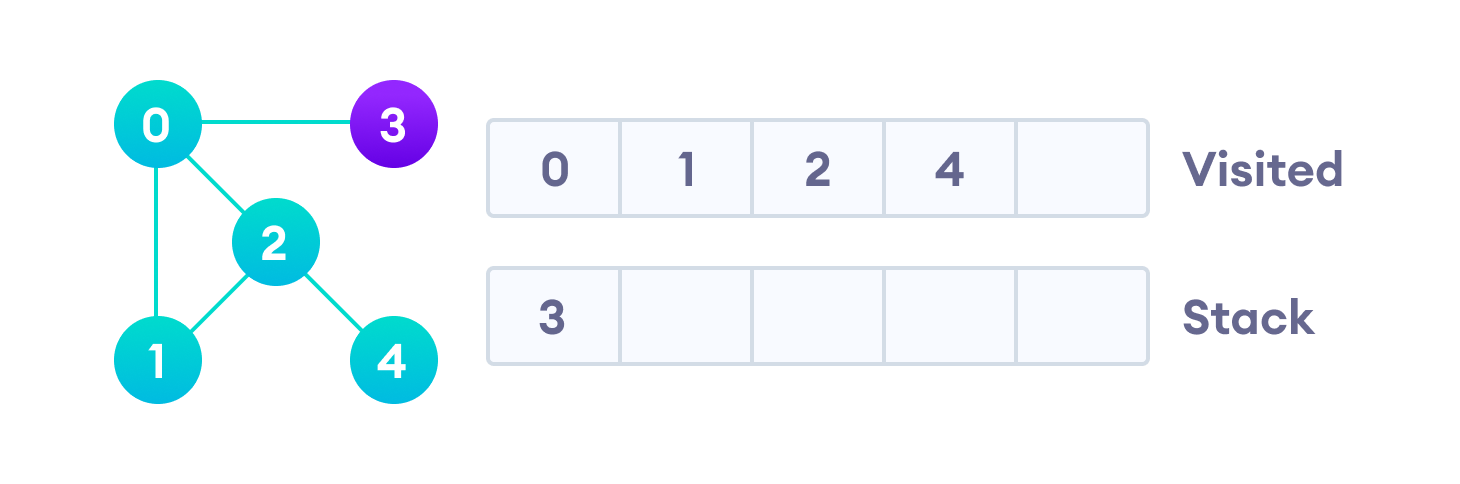
Начнем мы с вершины “0”. В первую очередь алгоритм поиска в глубину поместит ее саму в список “Пройденные” (на изображении “Visited”), а ее смежные вершины — в стек.

Выберите элемент (вершину) и поместите его в список “Пройденные”.

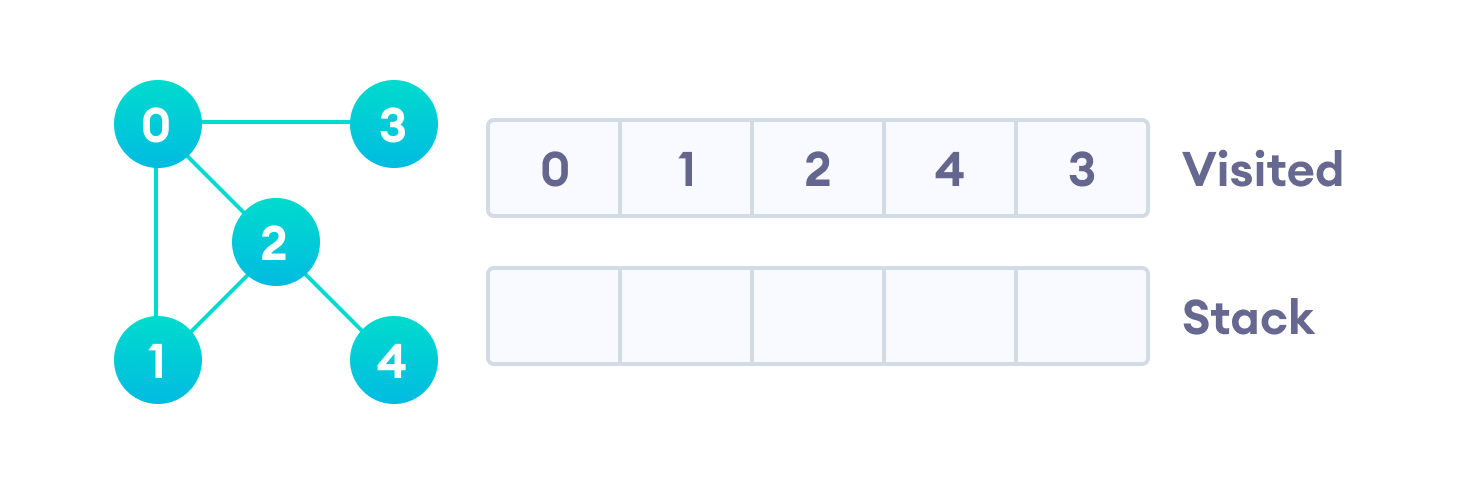
Затем мы берем следующий элемент сверху стека, т.е. к вершину “1”, и переходим к ее соседним вершинам. Поскольку вершина “0” уже пройдена, следующая вершина “2”.

Обход элемента на вершине стека.

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

Вершина “2” смежна непройденной вершине “4”, следовательно мы помещаем ее в верх стека.Добавляем вершину “4” в список “Пройденные” после прохождения.

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.

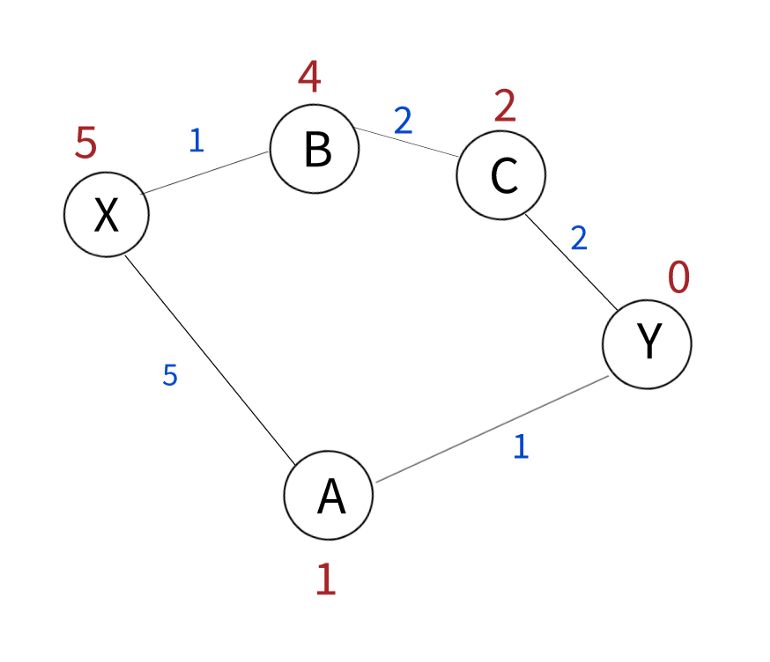
После проверки всех смежных вершин для вершины “3” стек остался пустым, а значит алгоритм обхода графа в глубину завершил свою работу.

**Пример реализации А\***

Говоря простым языком, алгоритм А\* находит оптимальный вариант благодаря вычислению суммарной стоимости всех путей между начальной и конечной точкой. Кстати, этот способ быстрее алгоритма Дейкстры благодаря эвристической функции.

* f(n) = g(n) + h(n)
* f(n): общая стоимость пути
* g(n): стоимость пути между текущей и начальной вершиной
* h(n): эвристическая функция

Разберем пример. У нас есть граф:



Допустим, мы хотим попасть из точки X в точку Y. Так как вершина Х не меняет своего положения, мы можем отбросить g(n) — ее значение равно 0. Эвристическое значение этой вершины выделено красным шрифтом — 5.

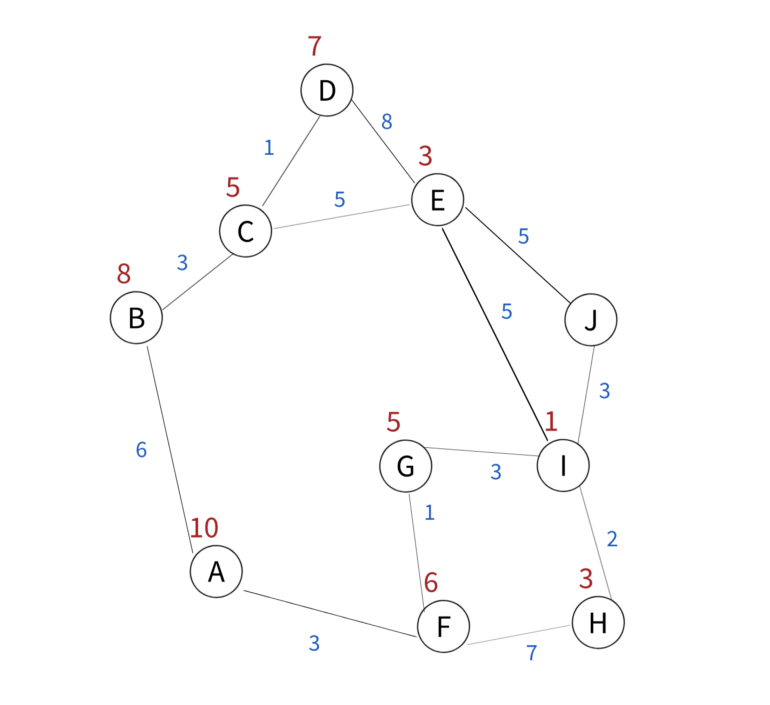
В подобных задачах эвристическое значение —  стоимость достижения рассматриваемой вершины из начальной.

Из вершины Х есть два пути.

Если мы перейдем в вершину А, g(n) будет равна 5 (стоимость пути), так как мы перемещаемся в новую вершину. Значение h(n) теперь равно 1. Значение f(n) в точке А будет равно 5+1 = 6. Теперь найдем значение f(n) каждой точки:

* X— A => g(A) + f(A) = 5 + 1 = 6,
* A — Y=> g(Y) + f(Y) = 6+ 0= 6,
* X— B => g(B) + f(B) = 1+ 4= 5,
* B — C => g(C) + f(C) = 3+ 2= 5,
* C — Y=> g(Y) + f(Y) = 5 + 0= 5,

Как видно из наших вычислений, кратчайший путь — X-B-C-Y. Его стоимость равна 5, в то время как X-A-Y — 6. С этим примером разобрались, рассмотрим следующий.



Допустим, мы хотим найти кратчайший путь от вершины А до вершины J.

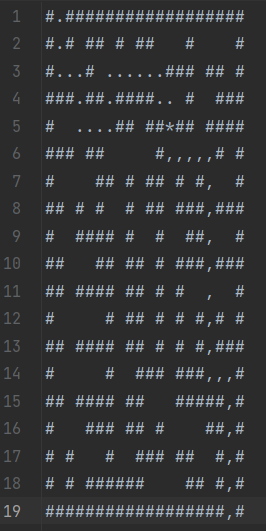
Из А есть только два пути — B и F. Вычислим стоимость: f(B) = 8 + 6 = 14 и f(F) = 3+6 =9. Следовательно, нам нужно перейти в вершину F — алгоритм продолжит работу отсюда.

Из точки F есть два пути — G и H. Снова вычислим стоимость:  f(G) = 4 +5 = 9 and f(H) = 10 + 3 = 13. Значит, мы переходим в точку G.

Следуя пути I—J, получаем следующее:  f(I) = 7 + 1 = 8 и f(J) = 10. Так как все значения, следующие за вершиной F, меньше f(B), возвращаться к вершине B не имеет смысла.

Но допустим другой вариант. Предположим, что f(I) больше f(B) при прохождении через F и G (f(I) > 14). В этом случае алгоритм A\* прекратит дальнейшую работу и переместится в вершину В. Но, так как f(C) > f(I), работа алгоритма продолжается именно в вершине I.

**Пример работы**



# Заключение

В ходе выполнения задания были изучены алгоритмы построения маршрута в графе, в частности алгоритмы поиска в глубину и А\*. Были выделены особенности реализации, необходимые в конкретной задаче поиска маршрута. В результате выполнения задания были сохранены результаты обходов лабиринта (от начала до ключа с помощью алгоритма поиска в глубину и от ключа до выхода с помощью алгоритма А\*) и получившиеся маршруты в файл. В целом, задание показало важность знаний алгоритмов поиска маршрута в графе и их практическое применение при решении конкретных задач.

# Список литературы

1. Алгоритм поиска A\* - Nuances of programming [Электронный ресурс]. URL: (<https://nuancesprog.ru/p/5572/>) (Дата обращения: 26.04.2023).

2. Алгоритм поиск A\*. Пошаговый разбор [Электронный ресурс]. URL: (<https://medium.com/nuances-of-programming/>) (Дата обращения: 26.04.2023).

3. Метод поиска пути в лабиринте при наличии помех [Электронный ресурс]. URL: (<https://cyberleninka.ru/>) (Дата обращения: 26.04.2023).

4.DFS (Depth-First Search) - поиск в глубину [Электронный ресурс]. URL: (<https://blog.skillfactory.ru/glossary/dfs/>) (Дата обращения: 26.04.2023).

5. Поиск в глубину (DFS) [Электронный ресурс]. URL: (<https://ikcprog.github.io/topics/dfs/>) (Дата обращения: 26.04.2023).

6. Обходы графов - Алгоритмика - Algorithmica [Электронный ресурс]. URL: (<https://algorithmica.org/ru/dfs>) (Дата обращения: 26.04.2023).

7.Алгоритм поиска в глубину - Информатика - Фоксфорд [Электронный ресурс]. URL: (<https://foxford.ru/wiki/informatika/algoritm-poiska-v-glubinu>) (Дата обращения: 26.04.2023).

# Приложение 1

## Листинг программы

import sys  
sys.setrecursionlimit(1500)  
  
def find\_start\_end\_points(maze):  
 for Y in range(len(maze[0])):  
 if maze[0][Y] == " ":  
 start = (0, Y)  
 break  
 for Y in range(len(maze[0])):  
 if maze[len(maze) - 1][Y] == " ":  
 end = (len(maze) - 1, Y)  
 break  
 return start, end  
  
  
def find\_key\_position(maze):  
 for i in range(len(maze)):  
 for j in range(len(maze[0])):  
 if maze[i][j] == "\*":  
 key = (i, j)  
 break  
 return key  
  
  
def dfs(maze, start, end, visited=None):  
 if visited is None:  
 visited = []  
  
 visited.append(start)  
  
 if start == end:  
 return [start]  
  
 directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  
  
 for direction in directions:  
 x, y = start[0] + direction[0], start[1] + direction[1]  
  
 if (0 <= x < len(maze) and 0 <= y < len(maze[x]) and maze[x][y] != "#" and (x, y) not in visited):  
 path\_found = dfs(maze, (x, y), end, visited)  
  
 if path\_found:  
 return [(start)] + path\_found  
  
 return None  
  
  
  
def a\_star(maze, start, end):  
 rows = len(maze)  
 cols = len(maze[0])  
  
 l = [[0 for i in range(cols)] for j in range(rows)]  
  
 l[start[0]][start[1]] = 1  
  
 check\_list = [(start[0], start[1], 0)]  
  
 while len(check\_list) > 0:  
 check\_list.sort(key=lambda x: x[2])  
 i, j, d = check\_list.pop(0)  
 if i == end[0] and j == end[1]:  
 path = []  
 while d != 0:  
 path.append((i, j))  
 if i > 0 and l[i - 1][j] == d - 1:  
 i, j, d = i - 1, j, d - 1  
 elif j > 0 and l[i][j - 1] == d - 1:  
 i, j, d = i, j - 1, d - 1  
 elif i < rows - 1 and l[i + 1][j] == d - 1:  
 i, j, d = i + 1, j, d - 1  
 elif j < cols - 1 and l[i][j + 1] == d - 1:  
 i, j, d = i, j + 1, d - 1  
 path.append((i, j))  
 path.reverse()  
 return path  
  
 if i > 0 and maze[i - 1][j] != "#" and l[i - 1][j] == 0:  
 check\_list.append((i - 1, j, d + 1))  
 l[i - 1][j] = d + 1  
  
 if j > 0 and maze[i][j - 1] != "#" and l[i][j - 1] == 0:  
 check\_list.append((i, j - 1, d + 1))  
 l[i][j - 1] = d + 1  
  
 if i < rows - 1 and maze[i + 1][j] != "#" and l[i + 1][j] == 0:  
 check\_list.append((i + 1, j, d + 1))  
 l[i + 1][j] = d + 1  
  
 if j < cols - 1 and maze[i][j + 1] != "#" and l[i][j + 1] == 0:  
 check\_list.append((i, j + 1, d + 1))  
 l[i][j + 1] = d + 1  
  
 return None  
  
  
def modify\_maze\_path(maze, pathToKey, pathToExit):  
 for coords in pathToKey:  
 x, y = coords  
 if maze[x][y] == " ":  
 maze[x][y] = "."  
 for coords in pathToExit:  
 x, y = coords  
 if maze[x][y] == " ":  
 maze[x][y] = ","  
 return maze  
  
  
def write\_modified\_maze\_to\_file(maze, file\_name):  
 with open(file\_name, 'w') as f:  
 for line in maze:  
 f.write("".join(line) + "\n")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 with open('MyLab.txt', 'r') as f:  
 maze = [list(line.strip()) for line in f.readlines()]  
  
 start, end = find\_start\_end\_points(maze)  
 key = find\_key\_position(maze)  
  
 pathToKey = dfs(maze, start, key)  
 pathToExit = a\_star(maze, key, end)  
 modified\_maze = modify\_maze\_path(maze, pathToKey, pathToExit)  
 write\_modified\_maze\_to\_file(modified\_maze, 'maze-for-me-done.txt')