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»

Выполнила студентка 1 курса гр. ИС-33 Марганова А.С.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

**Оглавление**

[Введение 3](#_Toc135785456)

[Задачи 3](#_Toc135785457)

[1. Теоретическая часть 4](#_Toc135785458)

[1.2 Алгоритм A\* 5](#_Toc135785459)

[2. Реализация алгоритма 7](#_Toc135785460)

[Пример работы 11](#_Toc135785461)

[Заключение 12](#_Toc135785462)

[Список литературы 13](#_Toc135785463)

[Листинг программы 14](#_Toc135785464)

# **Введение**

Алгоритмы поиска пути и структурное программирование являются важными аспектами в области компьютерных наук и информационных технологий. Они позволяют разработчикам создавать эффективные программные решения для решения различных задач, связанных с поиском оптимального пути и управлением данными.

Обход графа - это процесс прохождения по всем вершинам графа с целью найти определенную информацию или выполнить определенные операции. Существует несколько способов обхода графа, таких как обход в глубину (DFS) и обход в ширину (BFS). В обходе в глубину происходит переход к следующей вершине только после того, как были пройдены все соседние вершины текущей. Обход графа широко используется в алгоритмах поиска пути и решении других задач, связанных с графами.

Одним из алгоритмов является depth-first search, DFS (поиск в ширину)

**Цель работы:** реализовать алгоритмы обхода графа: depth-first search, DFS (поиск в глубину) и А\* для задачи поиска маршрута в лабиринте.

# **Задачи**

* Изучить алгоритмы построения маршрута в графе;
* 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**

**1.1. Алгоритм поиска в глубину**

Алгоритм поиска в глубину (DFS) является одним из основных алгоритмов обхода графа. Он использует принцип "погружения" в глубину графа до тех пор, пока не будет достигнут конечный узел или не будут обработаны все узлы.

Шаги алгоритма:

1. Выбрать начальный узел и пометить его как посещенный.

2. Перейти к не посещенному соседнему узлу.

3. Если все соседние узлы уже посещены, вернуться к предыдущему узлу.

4. Повторять шаги 2 и 3, пока не будут обработаны все узлы.

Рекурсивная реализация алгоритма:

1. Пометить текущий узел как посещенный.

2. Для каждого не посещенного соседнего узла вызвать рекурсивную функцию DFS.

3. Вернуться к предыдущему узлу.

Итеративная реализация алгоритма:

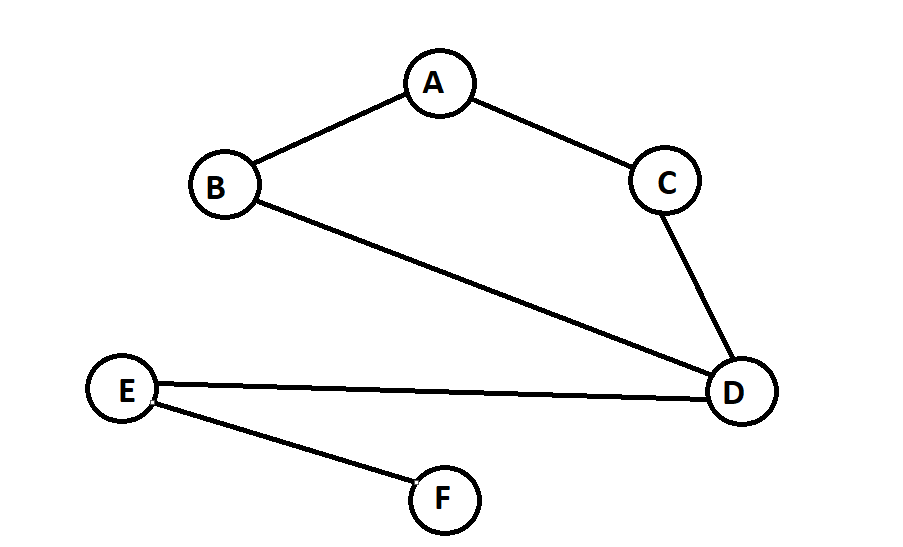
1. Создать стек и поместить в него начальный узел.

2. Пока стек не пуст, извлечь из него последний узел и пометить его как посещенный.

3. Для каждого не посещенного соседнего узла поместить его в стек.

4. Повторять шаги 2 и 3, пока стек не будет пуст.

Пример работы алгоритма на графе:



Рассмотрим граф с узлами A, B, C, D, E и F и ребрами (A,B), (A,C), (B,D), (C,D), (D,E) и (E,F).

Рекурсивная реализация:

1. Начинаем с узла A.

2. Помечаем A как посещенный и вызываем DFS для B.

3. Помечаем B как посещенный и вызываем DFS для D.

4. Помечаем D как посещенный и вызываем DFS для E.

5. Помечаем E как посещенный и вызываем DFS для F.

6. Возвращаемся к E, все его соседи уже посещены, возвращаемся к D.

7. Возвращаемся к B, все его соседи уже посещены, возвращаемся к A.

8. Вызываем DFS для C.

9. Помечаем C как посещенный и вызываем DFS для D.

10. Возвращаемся к C, все его соседи уже посещены, возвращаемся к A.

11. Все узлы обработаны.

Итеративная реализация:

1. Начинаем с узла A.

2. Помечаем A как посещенный и помещаем его в стек.

3. Извлекаем из стека A и для каждого не посещенного соседнего узла (B и C) помещаем его в стек и помечаем как посещенный.

4. Извлекаем из стека C и для не посещенного соседнего узла D помещаем его в стек и помечаем как посещенный.

5. Извлекаем из стека D и для не посещенного соседнего узла E помещаем его в стек и помечаем как посещенный.

6. Извлекаем из стека E и для не посещенного соседнего узла F помещаем его в стек и помечаем как посещенный.

7. Извлекаем из стека F, стек пуст.

8. Возвращаемся к E, все его соседи уже посещены, извлекаем из стека D.

9. Возвращаемся к D, все его соседи уже посещены, извлекаем из стека C.

10. Возвращаемся к C, все его соседи уже посещены, извлекаем из стека B.

11. Возвращаемся к B, все его соседи уже посещены, извлекаем из стека A.

12. Стек пуст, все узлы обработаны.

Сложность алгоритма поиска в глубину зависит от количества узлов и ребер графа. В худшем случае алгоритм посетит все узлы и ребра графа, что занимает O(V+E) времени, где V - количество узлов, E - количество ребер.

По сравнению с другими алгоритмами поиска, такими как алгоритм поиска в ширину или алгоритм Дейкстры, алгоритм поиска в глубину не находит кратчайший путь или оптимальный маршрут, но он может быть более эффективным для обхода больших графов или для поиска определенных узлов.

Алгоритм поиска в глубину может быть использован для поиска пути в графе, топологической сортировки и обхода дерева. Он также может быть расширен для работы с ограничением глубины, взвешенными графами и применен в машинном обучении и искусственном интеллекте.

* 1. **Алгоритм A\***

Алгоритм A\* (A-star) является эвристическим алгоритмом поиска кратчайшего пути в графе, который используется для нахождения оптимального пути между двумя точками. Алгоритм A\* основан на комбинации двух функций: оценки расстояния до цели и функции стоимости пути. Он использует эвристику, чтобы ускорить поиск и находить оптимальный путь.

Алгоритм A\* был разработан в 1968 году Петером Хартом, Нилом Коппелманом и Робертом Джонсоном в лаборатории искусственного интеллекта компании SRI International. Он был разработан как улучшение алгоритма Dijkstra, который был разработан в 1959 году Эдсгером Дейкстрой. Алгоритм A\* быстро стал популярным и нашел широкое применение в различных областях, включая игры, робототехнику, машинное обучение и другие.

Алгоритм A\* использует эвристику для оценки расстояния от текущей точки до целевой точки. Эта эвристика называется функцией h(x) и может быть реализована различными способами, например, как евклидово расстояние между двумя точками или как манхэттенское расстояние.

Алгоритм A\* также использует функцию стоимости пути g(x), которая оценивает стоимость пути от начальной точки до текущей точки. Эта функция может быть определена как расстояние между двумя точками или как сумма весов ребер, соединяющих эти точки.

Алгоритм A\* выбирает наилучший путь, используя комбинацию функций h(x) и g(x). Он выбирает следующую точку для посещения, которая имеет наименьшую сумму функций h(x) и g(x). Это позволяет алгоритму A\* находить оптимальный путь между начальной и целевой точками.

Псевдокод алгоритма:

1. Инициализировать список открытых вершин

2. Инициализировать список закрытых вершин

3. Добавить начальную вершину в список открытых вершин

4. Пока список открытых вершин не пуст:

- Выбрать вершину с наименьшей суммой функций h(x) и g(x)

- Если эта вершина является целевой, то завершить поиск

- Иначе добавить эту вершину в список закрытых вершин и найти все соседние вершины

- Для каждой соседней вершины:

- Если эта вершина уже находится в списке закрытых вершин, то пропустить ее

- Иначе, если эта вершина еще не находится в списке открытых вершин, то добавить ее в список и вычислить функции h(x) и g(x)

- Иначе, если новый путь через эту вершину короче, чем старый путь, то обновить функции h(x) и g(x) для этой вершины

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

1. **Реализация алгоритма**

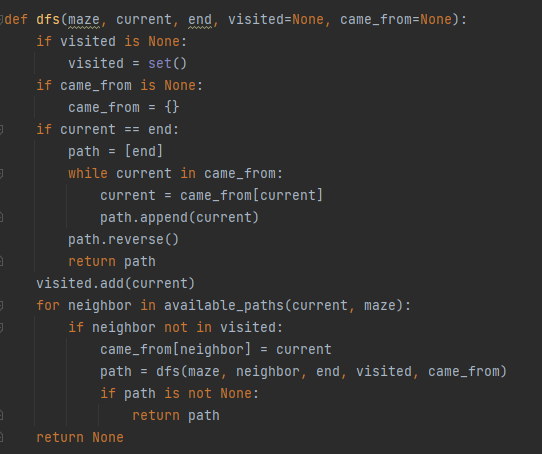
**Реализация алгоритма поиска в глубину в данной программе:**

Входными параметрами являются лабиринт (maze), начальная точка (current) и конечная точка (end). Также имеются два необязательных параметра: множество посещенных точек (visited) и словарь, который хранит информацию о том, из какой точки пришли в указанную точку (came\_from).

Функция начинает с проверки, была ли создана множество посещенных точек и словарь came\_from. Если нет, то они создаются. Затем происходит проверка, является ли текущая точка конечной. Если да, то мы начинаем следовать по цепи родительских точек, пока не достигнем начальной точки. После этого список пути обращается и возвращается.

Если текущая точка не является конечной, она добавляется в множество посещенных точек, и происходит обход всех доступных путей из точки current. Если соседняя точка не была посещена, то она добавляется в словарь came\_from и рекурсивно вызывается функция dfs со следующей точкой в качестве нового текущего местоположения.

Если на следующей точке не найден путь к конечной точке, функция возвращает пустой результат. Если путь найден, он сразу же возвращается функцией.



**Реализация алгоритма A\* в данной программе:**

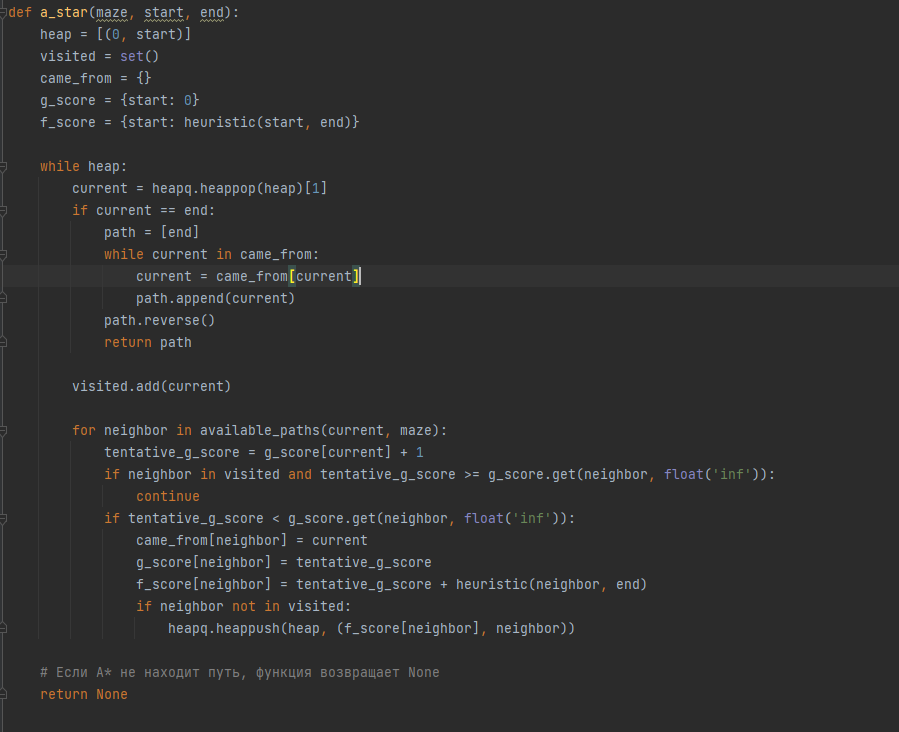
Функция a\_star реализует алгоритм поиска пути A\* для двумерного лабиринта, заданного в виде матрицы maze. Функция принимает координаты начальной точки start и конечной точки end.

Перед началом выполнения алгоритма создаются необходимые структуры данных: heap – это куча, в которой хранятся вершины, доступные для рассмотрения, visited – это множество вершин, которые уже были рассмотрены, came\_from – это словарь, в котором для каждой вершины хранится ссылка на вершину, из которой пришли, g\_score – это словарь, в котором для каждой вершины хранится длина кратчайшего пути от начальной вершины, f\_score – это словарь, в котором для каждой вершины хранится оценка стоимости пути от начальной вершины до конечной.

Далее выполняется главный цикл алгоритма, пока heap не пуст. Из heap извлекается вершина с минимальной оценкой f\_score, которая становится текущей вершиной. Если текущая вершина совпадает с конечной вершиной, то восстанавливается путь до этой вершины, обратным проходом по came\_from.

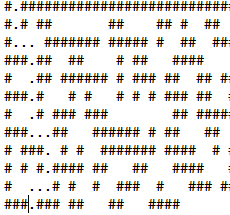
Для каждой рассматриваемой вершины выполняется перебор всех ее соседей, доступных для перехода. Для каждого соседа вычисляется предварительная оценка длины пути tentative\_g\_score, которая получается из длины пути от начальной вершины до текущей вершины g\_score[current] плюс стоимость перехода до соседа, в данном случае стоимость перехода равна единице. Если сосед уже был рассмотрен и предварительная оценка tentative\_g\_score получилась больше текущей длины пути g\_score[neighbor], то этот сосед пропускается и рассматривается следующий. Если предварительная оценка tentative\_g\_score получилась меньше текущей длины пути g\_score[neighbor], то обновляются значения для came\_from, g\_score и f\_score для этого соседа. Если же сосед еще не был рассмотрен, то его добавляют в heap с оценкой f\_score[neighbor], равной сумме tentative\_g\_score и эвристической оценки стоимости пути от этого соседа до конечной вершины.

Если алгоритм не нашел путь до конечной вершины, то функция возвращает None.

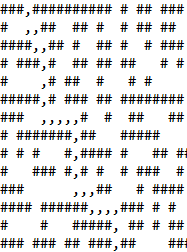


# **Пример работы**

**Поиск в глубину**



**Алгоритм А\***



# **Заключение**

В рамках выполнения задачи были разработан программный код для решения задачи прохождения маршрута в лабиринте. Для этого были использованы алгоритмы поиска в глубину и A\*. Алгоритм А\* и алгоритм поиска в глубину являются важными инструментами в области искусственного интеллекта и компьютерных наук. Оба алгоритма используются для решения задач поиска оптимального пути в графах и имеют свои преимущества и недостатки. Алгоритм А\* обладает более высокой эффективностью и точностью, но требует большего объема вычислений. Алгоритм поиска в глубину, в свою очередь, более прост в реализации и требует меньше вычислительных ресурсов, но может не дать оптимального решения. В результате работы был создан код для прохождения маршрута от начальной координаты аватара до ключа с помощью алгоритма поиска в глубину, а затем - от ключа до ближайшего выхода с помощью A\*.

# 

# **Список литературы**

1. Кормен Т., Лейзерсон Ч., Ривест Р. Глава 22. Элементарные алгоритмы для работы с графами // Алгоритмы: построение и анализ(второе издание). — М.: «Вильямс», 2005. — С. 622—632.
2. Левитин А. В. Глава 5. Метод уменьшения размера задачи: Поиск в глубину // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 212–215. — 576 с. — ISBN 978-5-8459-0987-9
3. Рассел, С., Норвиг, П. (2009). Artificial Intelligence: A Modern Approach (3-е изд.). Москва: ДМК Пресс. (Дата обращения: 20.05.2023).
4. vuzlit.com [Электронный ресурс] / URL: <https://vuzlit.com/957009/poisk_glubinu> (дата обращения: 20.05.2023)
5. Харт П., Нилсон Н., Раффо Д. Эффективный алгоритм поиска кратчайшего пути // Искусственный интеллект. – 1968. – Т. 1. – № 3. – С. 189-208.

# **Листинг программы**

import heapq  
  
def available\_paths(coordsXY, maze):  
 LenMazeY = len(maze[0])  
 LenMazeX = len(maze)  
 coordsX = coordsXY[0]  
 coordsY = coordsXY[1]  
 possibleWays = []  
  
 if (coordsX - 1) >= 0 and maze[coordsX - 1][coordsY] == " ": #Север  
 coord\_for\_append = (coordsX - 1, coordsY)  
 possibleWays.append(coord\_for\_append)  
  
 if (coordsY + 1) < LenMazeY and maze[coordsX][coordsY + 1] == " ": #Восток  
 coord\_for\_append = (coordsX, coordsY + 1)  
 possibleWays.append(coord\_for\_append)  
  
 if (coordsX + 1) < LenMazeX and maze[coordsX + 1][coordsY] == " ": #Юг  
 coord\_for\_append = (coordsX + 1, coordsY)  
 possibleWays.append(coord\_for\_append)  
  
 if (coordsY - 1) >= 0 and maze[coordsX][coordsY - 1] == " ": #Запад  
 coord\_for\_append = (coordsX, coordsY - 1)  
 possibleWays.append(coord\_for\_append)  
 return possibleWays  
  
  
def heuristic(current, end):  
 return abs(current[0] - end[0]) + abs(current[1] - end[1])  
  
  
def dfs(maze, current, end, visited=None, came\_from=None):  
 if visited is None:  
 visited = set()  
 if came\_from is None:  
 came\_from = {}  
 if current == end:  
 path = [end]  
 while current in came\_from:  
 current = came\_from[current]  
 path.append(current)  
 path.reverse()  
 return path  
 visited.add(current)  
 for neighbor in available\_paths(current, maze):  
 if neighbor not in visited:  
 came\_from[neighbor] = current  
 path = dfs(maze, neighbor, end, visited, came\_from)  
 if path is not None:  
 return path  
 return None  
  
  
def a\_star(maze, start, end):  
 heap = [(0, start)]  
 visited = set()  
 came\_from = {}  
 g\_score = {start: 0}  
 f\_score = {start: heuristic(start, end)}  
  
 while heap:  
 current = heapq.heappop(heap)[1]  
 if current == end:  
 path = [end]  
 while current in came\_from:  
 current = came\_from[current]  
 path.append(current)  
 path.reverse()  
 return path  
  
 visited.add(current)  
  
 for neighbor in available\_paths(current, maze):  
 tentative\_g\_score = g\_score[current] + 1  
 if neighbor in visited and tentative\_g\_score >= g\_score.get(neighbor, float('inf')):  
 continue  
 if tentative\_g\_score < g\_score.get(neighbor, float('inf')):  
 came\_from[neighbor] = current  
 g\_score[neighbor] = tentative\_g\_score  
 f\_score[neighbor] = tentative\_g\_score + heuristic(neighbor, end)  
 if neighbor not in visited:  
 heapq.heappush(heap, (f\_score[neighbor], neighbor))  
  
 # Если A\* не находит путь, функция возвращает None  
 return None  
  
  
with open('maze-for-u.txt', 'r') as f:  
 maze = [list(line.strip()) for line in f.readlines()]  
  
start = None  
end = None  
key = None  
  
for i in range(len(maze)):  
 for j in range(len(maze[0])):  
 if maze[i][j] == "\*":  
 key = (i, j)  
 maze[i][j] = " "  
 break  
 if key is not None:  
 break  
  
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  
  
if key is None:  
 print("Key not found in the maze!")  
 exit()  
  
pathToKey = dfs(maze, start, key)  
if pathToKey is None:  
 print("There is no path to the key!")  
 exit()  
  
pathToExit = a\_star(maze, key, end)  
if pathToExit is None:  
 print("There is no path to the exit!")  
 exit()  
  
  
#от точки-ключа до выхода "."  
for coords in pathToKey:  
 x, y = coords  
 maze[x][y] = "."  
  
#от точки-ключа до выхода ","  
for coords in pathToExit:  
 x, y = coords  
 maze[x][y] = ","  
  
# Записываем измененный лабиринт в файл  
with open('maze-for-me-done.txt', 'w') as f:  
 for line in maze:  
 f.write("".join(line) + "\n")