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 Турков Е.М.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

# Содержание

[Содержание 2](#_Toc137107315)

[Введение 3](#_Toc137107316)

[Цель: 3](#_Toc137107317)

[Задачи: 3](#_Toc137107318)

I.[Теоретическая часть 4](#_Toc137107319)

[Графы: 4](#_Toc137107320)

[Обход графа: 4](#_Toc137107321)

[Алгоритм поиска в глубину: 5](#_Toc137107322)

[Алгоритм A\*: 5](#_Toc137107323)

II.[Реализация алгоритма 7](#_Toc137107324)

III.[Пример работы 9](#_Toc137107325)

IV.[Заключение 11](#_Toc137107326)

[Источники 12](#_Toc137107327)

[Приложение 1 13](#_Toc137107328)

# Введение

Алгоритмы обхода графа — это тип алгоритма, который используется для обхода графа, который представляет собой набор узлов и ребер, соединенных вместе. Этот тип алгоритма используется для поиска определенного узла или пути в графе. Алгоритмы обхода графа реализованы в программировании, чтобы помочь решить такие проблемы, как поиск кратчайшего пути между двумя узлами, обнаружение циклов в графе или выполнение топологической сортировки. Кроме того, алгоритмы обхода графа можно использовать для поиска пути между двумя узлами, который должен проходить через определенное количество других узлов, или для определения того, какие узлы связаны друг с другом.

Алгоритмы обхода графа могут быть реализованы на различных языках программирования, таких как Java, C++ и Python. Это делает их широко применимыми к различным типам проблем и позволяет найти эффективные и надежные решения. [4]

## Цель:

Написать алгоритмы поиска в глубину и A\* для поиска выхода из лабиринта и сокровища в произвольной точке лабиринта.

## Задачи:

* Изучить работу алгоритмов движения по графу и способы их реализации;
* Подготовить необходимые исходные данные;
* Реализовать поиск ключа с помощью алгоритма поиска в глубину;
* Реализовать алгоритм A\*для поиска выхода из лабиринта;
* Сохранить получившийся лабиринт с отмеченными путями, как отдельный файл.

# Теоретическая часть

## Графы:

Граф представляет собой абстрактную структуру данных, которая состоит из вершин (заключенных в узлы) и ребер, соединяющих эти вершины. Графы широко используются в информатике, математике, социологии, биологии, и других областях, где необходима визуализация и анализ отношений между объектами.

Существует несколько видов графов, включая простые графы, мультиграфы, ориентированные графы и взвешенные графы.

Простой граф является наиболее распространенным типом. Он представляет собой граф, в котором между вершинами может быть только одно ребро. Мультиграфы же позволяют наличие нескольких ребер между одной и той же парой вершин. Ориентированный граф имеет направленные ребра, то есть связи между вершинами имеют направление. И, наконец, взвешенный граф содержит не только ребра, но и численное значение (вес), которое описывает характер связи между вершинами. [3]

## Обход графа:

Обход графа - это процесс посещения (и иногда обработки) каждой вершины в графе. Существует несколько способов обхода графов, включая поиск в глубину (DFS), поиск в ширину (BFS) и топологический поиск.

* При поиске в глубину для обхода графа сначала посещается текущая вершина, затем одна из соседних вершин и так далее пока не достигнута конечная вершина или пока не обойдены все соседние вершины. Если какая-то вершина была уже посещена, то процесс переходит к следующей вершине. Поиск в глубину наиболее эффективен для обхода неориентированных графов.
* При поиске в ширину сначала посещаются все соседние вершины первого уровня, затем все соседние вершины второго уровня, и так далее, пока не достигнута конечная вершина или пока не обойдены все вершины. Поиск в ширину наиболее эффективен для обхода ориентированных графов.
* Топологический поиск используется для графов, в которых есть направление ребер. В этом случае вершины упорядочиваются таким образом, чтобы любое направленное ребро всегда идет от вершины с меньшим индексом к большему. Топологический поиск используется в алгоритмах, связанных с задачами планирования. [5]

## Алгоритм поиска в глубину:

Поиск в глубину (DFS) — это тип алгоритма обхода графа. Он работает, начиная с корневого узла дерева и исследуя как можно дальше каждую ветвь, прежде чем возвращаться назад. Алгоритм использует стек в качестве структуры данных, чтобы отслеживать, какие узлы исследовать дальше.

Основная идея DFS проста: начиная с одного узла, сначала исследуйте все его соседи. Если один из соседей ранее не посещался, исследуйте его соседей таким же образом. Продолжайте этот процесс, пока не будут посещены все подключенные узлы. Ключевым моментом DFS является то, что он позволяет избежать повторного посещения одного и того же узла.

Алгоритм работы следующий:

1. Начните с заданного узла графа.

2. Проверьте, посещался ли узел ранее. Если нет, отметьте его как посещенный и добавьте в стек.

3. Исследуйте каждого из соседей узла. Если они не были посещены, добавьте их в стек.

4. Когда все соседи текущего узла посещены, извлеките узел из стека и перейдите к следующему.

5. Повторяйте шаги 2–4, пока стек не станет пустым.

Временная сложность DFS равна O (V + E), где V — количество вершин, а E — количество ребер. Это делает его подходящим для приложений обхода графа, где количество ребер намного больше, чем количество вершин.

DFS можно использовать для поиска связанных компонентов, топологической сортировки, обнаружения циклов в графе и многого другого. Его можно использовать для решения многих задач, связанных с графами, таких как поиск кратчайшего пути между двумя узлами или поиск остовного дерева графа. Его также можно использовать для решения головоломок и лабиринтов.[1][2]

## Алгоритм A\*:

A\* (A-Star) — алгоритм информированного поиска, используемый для поиска пути и обхода графа. Это расширение алгоритма Дейкстры, добавляющее к поиску эвристику. A\* способен найти наиболее оптимальный путь к целевому узлу от начального узла.

Алгоритм A\* работает, поддерживая два набора узлов, называемых открытыми и закрытыми наборами, которые представляют узлы, которые необходимо рассмотреть, и уже рассмотренные узлы, соответственно. На каждом шаге алгоритм рассматривает узел в открытом множестве с наименьшей оценочной стоимостью. Эта стоимость оценивается как сумма фактической стоимости от начального узла до текущего узла плюс предполагаемая стоимость от текущего узла до цели.

Затем алгоритм добавляет этот узел в закрытый набор и расширяет его соседей, добавляя всех непосещенных соседей в открытый набор. Этот процесс повторяется до тех пор, пока либо не будет достигнут целевой узел, либо к открытому множеству нельзя будет добавить больше узлов. Если путь к цели найден, алгоритм возвращает путь от начала до цели.

Эвристика, используемая в алгоритме A\*, представляет собой оценку стоимости перехода от текущего узла к цели. Эта оценка обычно получается из эвристической функции для конкретной задачи, которая учитывает определенные свойства пространства поиска.

Эвристика должна быть допустимой, а это означает, что она никогда не переоценивает стоимость достижения цели. Если эвристика последовательна, она также никогда не должна недооценивать стоимость достижения цели. Если эвристика одновременно допустима и непротиворечива, она называется монотонной, и алгоритм A\* гарантированно найдет оптимальный путь.

Алгоритм A\* — эффективный метод поиска пути, но он не лишен недостатков. Это требует больших вычислительных ресурсов и может стать неэффективным, если эвристическая функция недооценивает стоимость достижения цели. Кроме того, важно выбрать соответствующую эвристическую функцию, поскольку алгоритм хорош настолько, насколько хороша эвристика.

Несмотря на эти недостатки, алгоритм A\* широко используется во многих приложениях, таких как робототехника, компьютерные игры и картографические сервисы в Интернете. Было доказано, что это эффективный и действенный метод поиска пути при правильном использовании. [1][2]

# Реализация алгоритма

**Реализация алгоритма BFS:**

В нашем случае лабиринт можно рассматривать как двумерную сетку узлов или вершин, где каждая ячейка сетки представляет собой одну вершину. Каждая вершина соединена с соседними вершинами (вверх, вниз, влево, вправо), если они не являются стеной ('#') в лабиринте. Начальная позиция — это начальная вершина, с которой мы начинаем поиск, а «конечная» позиция — это целевая вершина (точка выхода из лабиринта).

Алгоритм DFS начинается с добавления начальной позиции в стек. В каждой итерации цикла алгоритм удаляет последний элемент стека, представляющий текущую обрабатываемую вершину или узел, и проверяет, является ли он «конечной» позицией. Если так, он возвращает путь для достижения этой позиции.

Если текущая обрабатываемая вершина не является «конечной» позицией, то алгоритм помечает вершину как посещенную и расширяет ее окрестности (вверх, вниз, влево и вправо). Затем алгоритм выбирает одного из соседей для дальнейшего изучения. В типичной реализации DFS алгоритм выбирает первого непосещенного соседа в предопределенном порядке, таком как порядок направлений по часовой стрелке.

Когда найден непосещенный сосед, алгоритм добавляет его в стек вместе с путем до достижения этой вершины. Затем алгоритм повторяет процесс, удаляя последний элемент из стека (который является самым последним добавленным соседом).

Алгоритм DFS продолжает исследование по одному пути, пока не зайдет в тупик. Когда тупик достигнут, алгоритм возвращается к самому последнему узлу, у которого есть неисследованные соседи, а затем продолжает исследовать следующего непосещенного соседа этого узла.

Таким образом, алгоритм DFS работает, исследуя как можно дальше каждую ветвь дерева поиска, прежде чем вернуться к самому последнему неисследованному узлу. В нашем случае он используется для обхода и поиска всех возможных путей в лабиринте, пока не будет найден путь к выходу.

Анализ алгоритма DFS показывает, что он имеет временную и пространственную сложность O(b^m), где b — коэффициент ветвления лабиринта (т. е. количество соседей вершины), а m — максимальная глубина лабиринта.

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.

**Реализация алгоритма A\* (A-star):**

* + 1. Создайте пустой открытый список (open list) и закрытый список (closed list).
    2. Добавьте начальную вершину в открытый список.
    3. Пока открытый список не пуст, выполните следующие действия:

1. Выберите вершину с наименьшей оценкой F из открытого списка. Это будет текущая вершина.
2. Если текущая вершина является целевой вершиной, вы нашли путь.
3. Переместите текущую вершину из открытого списка в закрытый список.
   * Для каждой соседней вершины текущей вершины выполните следующие действия:
     1. Если соседняя вершина уже находится в закрытом списке, проигнорируйте ее.
     2. Если соседняя вершина еще не находится в открытом списке, добавьте ее туда и вычислите оценку F.
     3. Если соседняя вершина уже находится в открытом списке, обновите ее оценку F, если новый путь короче.
4. Если открытый список стал пустым, и вы так и не нашли целевую вершину, значит путь не существует.

# Пример работы

Ниже приложены скриншоты результатов работы программного кода. Символы “#” обозначают стены, а пробелы между ними – пространство, по которому можно ходить. Входом в лабиринт является первая пуста верхняя левая клетка, отсюда начинается построение маршрута при помощи жадного алгоритма. Сам маршрут от начала до сокровища «\*» помечается «.». После подбора сокровища, оно пометится символом «;» (рис. 1)

По достижению сокровища «\*» маршрут до выхода начинает строится с помощью алгоритма А\* и помечается «,».(рис. 2)



Рис(1) – путь от старта до ключа(“.”)



Рис(2) – путь от ключа к выходу (“,”)

# Заключение

В заключение, эта курсовая работа представляет собой попытку рассмотреть два разных алгоритма для решения одной и той же проблемы – поиска пути. Реализация этих алгоритмов на языке Python с помощью структур данных и графической библиотеки позволит получить полную картину о том, как работают эти методы и какие их достоинства и недостатки.

В результате проделанной работы был создан и реализован алгоритм генерации путей в графе с использованием алгоритма поиска в глубину и алгоритма A\*. Для разработки программы потребовались исследования в области обработки графов, алгоритмов обхода графов и ранжирования приоритетов.

В рамках нашего готового продукта мы успешно реализовали и применили оба этих алгоритма для решения задачи поиска пути. (**приложение 1**)

Алгоритмы поиска пути играют ключевую роль во многих областях и приложениях. Они предлагают различные подходы к нахождению пути и имеют свои преимущества и ограничения. Выбор конкретного алгоритма зависит от требований задачи, характеристик графа и контекста применения.

# Литература и источники

1. Кормен, Т., & Лейзерсон, Ч. (2009). Алгоритмы: построение и анализ.
2. Алгоритмы: построение и анализ: учебник / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — 2-е изд. — М.: Вильямс, 2006. — 1296 с.
3. Кнут, Д. (2011). Искусство программирования. Том 1: Основные алгоритмы.
4. Тарьян, Р. (1972). Графы. Addison-Wesley.
5. Хопкрофт, Дж., Мотвани, Р., & Ульман, Д. (2007). Введение в теорию автоматов, языков и вычислений.
6. Боллобаш, Б. (1998). Modern graph theory (Vol. 184).
7. Depth First Search or DFS for a Graph. [Электронный ресурс] URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/(Дата обращения: 30.05.2023)
8. 7."A\* Search Algorithm", GeeksforGeeks, [Электронный ресурс] URL: https://www.geeksforgeeks.org/a-search-algorithm/ (Дата обращения: 30.05.2023)

# Приложение 1

from collections import deque

#1. проверка доступности пути и шаги

def possible\_ways(coordsXY, maze):

LenMazeY = len(maze[0])

LenMazeX = len(maze)

coordsX = coordsXY[0]

coordsY = coordsXY[1]

free\_way = []

if (coordsX - 1) >= 0 and maze[coordsX - 1][coordsY] == " ":

coord\_for\_append = (coordsX - 1, coordsY)

free\_way.append(coord\_for\_append)

if (coordsX + 1) < LenMazeX and maze[coordsX + 1][coordsY] == " ":

coord\_for\_append = (coordsX + 1, coordsY)

free\_way.append(coord\_for\_append)

if (coordsY - 1) >= 0 and maze[coordsX][coordsY - 1] == " ":

coord\_for\_append = (coordsX, coordsY - 1)

free\_way.append(coord\_for\_append)

if (coordsY + 1) < LenMazeY and maze[coordsX][coordsY + 1] == " ":

coord\_for\_append = (coordsX, coordsY + 1)

free\_way.append(coord\_for\_append)

return free\_way

#2. A\*

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

open\_list = [start]

closed\_list = []

came\_from = {}

g\_score = {start: 0}

f\_score = {start: heuristic(start, end)}

while open\_list:

current = min(open\_list, key=lambda x: f\_score[x])

if current == end:

path = [end]

while current in came\_from:

current = came\_from[current]

path.append(current)

path.reverse()

return path

open\_list.remove(current)

closed\_list.append(current)

for neighbor in possible\_ways(current, maze):

if neighbor in closed\_list:

continue

tentative\_g\_score = g\_score[current] + 1

if neighbor not in open\_list or tentative\_g\_score < g\_score[neighbor]:

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = tentative\_g\_score + heuristic(neighbor, end)

if neighbor not in open\_list:

open\_list.append(neighbor)

return open\_list

#3. deep search

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

def dfs(maze, start, end):

stack = [(start, [start])]

visited = set()

while stack:

current, path = stack.pop()

if current == end:

return path

visited.add(current)

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

if neighbor not in visited:

stack.append((neighbor, path + [neighbor]))

return None

#4. определение входа, выхода, ключа

def heuristic(current, end):

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

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

maze = [list(line.strip()) for line in f.readlines()]

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

for i in range(len(maze)):

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

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

key = (i, j)

maze[i][j] = " "

break

#5. решение лабиринта

pathToKey = dfs(maze, start, key)

pathToExit = a\_star(maze, key, end)

for coords in pathToKey:

x, y = coords

maze[x][y] = "."

for coords in pathToExit:

x, y = coords

if maze[x][y] == ".":

maze[x][y] = ";"

else:

maze[x][y] = ","

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

for line in maze:

f.write("".join(line) + "\n")