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: «Реализация алгоритмов поиска пути в лабиринте»

Выполнил студент 1 курса гр. ИС-33 Варнашова Д.А.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Содержание 2](#_Toc136506334)

[Введение 3](#_Toc136506335)

[Задачи 3](#_Toc136506336)

[1.Теоретическая часть 4](#_Toc136506337)

[1.1 Алгоритм Дейкстры 4](#_Toc136506338)

[1.2 Алгоритм A\* 4](#_Toc136506339)

[1.3 Алгоритм поиска в ширину 5](#_Toc136506340)

[1.4 Алгоритм поиска в глубину 7](#_Toc136506341)

[1.5 Жадный алгоритм 8](#_Toc136506342)

[2. Реализация алгоритма 9](#_Toc136506343)

[3. Пример работы 11](#_Toc136506344)

[Заключение 12](#_Toc136506345)

[Список литературы 13](#_Toc136506346)

[Приложение 1 14](#_Toc136506347)

[Листинг программы 14](#_Toc136506348)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании. Существует множество алгоритмов обхода графа, при своей работе все они отличаются скоростью, точностью и многими другими параметрами.

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

# Задачи

* Изучить алгоритмы построения маршрута в графе;
* Выделить особенности реализации, необходимые в конкретной задаче поиска маршрута;
* Подготовить исходные данные: лабиринт, координаты точек для посещения при обходе;
* Написать программу, которая использует выбранный алгоритм для нахождения маршрута и сохраняет результаты в файл
* Протестировать программу на различных наборах исходных данных и убедиться в ее правильной работе.

# 1.Теоретическая часть

## 1.1 Алгоритм Дейкстры

Алгоритм Дейкстры - это алгоритм поиска кратчайшего пути в графе с неотрицательными весами ребер. Он основан на идее постепенного расширения дерева кратчайших путей из начальной точки до всех остальных вершин.

Теория алгоритма Дейкстры основана на понятии "метки вершины". Начальная вершина помечается нулевым расстоянием, а все остальные - бесконечностью. Затем алгоритм выбирает вершину с наименьшей меткой, добавляет ее в дерево кратчайших путей и обновляет метки всех смежных с ней вершин. Если новая метка короче, чем старая, она обновляется.

Алгоритм продолжает выполняться, пока все вершины не будут помечены. В конечном итоге, расстояние от начальной вершины до любой другой вершины будет равно значению ее метки.

Алгоритм Дейкстры имеет полиномиальную сложность времени O(|E|+|V|log|V|), где |E| - количество ребер, а |V| - количество вершин в графе. Однако, это возможно только при использовании эффективных структур данных, таких как куча (heap), для выбора вершины с наименьшей меткой.

Важно отметить, что алгоритм Дейкстры работает только для графов с неотрицательными весами. Если граф содержит ребра с отрицательными весами, необходимо использовать другой алгоритм, например, алгоритм Беллмана-Форда.

## 1.2 Алгоритм A\*

Алгоритм A\* — это эвристический алгоритм поиска кратчайшего пути в графе с весами ребер. Он использует оценку расстояния от текущей вершины до целевой вершины для выбора следующей вершины в пути и достигает более высокой производительности, чем алгоритм Дейкстры, за счет использования эвристической оценки [1].

Алгоритм A\* работает похожим образом на алгоритм Дейкстры, но добавляет эвристическую функцию, которая оценивает расстояние от текущей вершины до целевой вершины. Эта функция, известная как функция оценки стоимости, помогает алгоритму ориентироваться в нужном направлении и выбирать более перспективные пути.

Одной из самых распространенных функций оценки стоимости является евклидово расстояние между текущей вершиной и целевой вершиной. Однако, в зависимости от конкретной задачи, могут быть использованы и другие функции оценки.

Алгоритм A\* является оптимальным, если используется эвристическая функция, удовлетворяющая условию "эвристика никогда не переоценивает действительную стоимость пути". Это означает, что эвристическая функция должна быть допустимой.

Алгоритм A\* широко используется в робототехнике, видеоиграх, планировании маршрутов и других областях, где требуется нахождение оптимальных путей в графах [2].

## 1.3 Алгоритм поиска в ширину

Алгоритм поиска в ширину был разработан независимо Муром и Ли для разных приложений (поиск пути в лабиринте и разводка проводников соответственно) в 1959 и 1961 годах [5].

Алгоритм поиска в ширину заключается в том, чтобы разведать вершины графа послойно, в порядке увеличения расстояния от стартовой ноды.

Рассмотрим на примере и возьмем для этого простой связный ненаправленный граф, ребра которого не имеют весов или временных меток. Слой 0 содержит стартовую вершину. Слой 1 будет содержать множество вершин, которые находятся на расстояние одного ребра от стартовой. Каждый последующий слой будет удаляться от стартовой вершины ровно на одно ребро. Алгоритм разведает сначала все ближайшие к стартовой ноды, затем более удаленные и т.д. и завершит работу, когда будут разведаны все вершины и алгоритм не сможет пройти дальше [6].

Этот алгоритм можно сравнить с поджиганием соседних вершин графа: сначала мы зажигаем одну вершину (ту, из которой начинаем путь), а затем огонь за один элементарный промежуток времени перекидывается на все соседние с ней не горящие вершины. В последствие то же происходит со всеми подожженными вершинами. Таким образом, огонь распространяется «в ширину». В результате его работы будет найден кратчайший путь до нужной клетки [5].

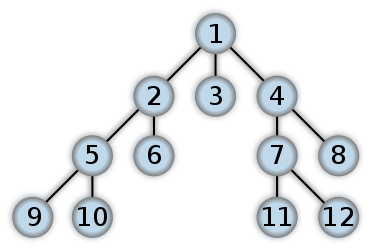
Алгоритм реализуется на основе очереди FIFO (First In, First Out – «Первым пришёл — первым ушёл»), с помощью которого отслеживаются ноды, которые алгоритм уже посещал. Очередь позволяет добавлять объекты в конец списка и удалять объекты из начала за постоянное время.

Рисунок 1 – Поиск в ширину

## 1.4 Алгоритм поиска в глубину

Алгоритм поиска в глубину отличается от поиска в ширину более агрессивным продвижением по графу [6]. 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.

Как и BFS, DFS помечает ноду каждый раз, как ее обнаруживает. На каждой итерации алгоритм обходит в произвольном порядке ноды, ближайшие к текущей. На первой же найденной вершине алгоритм будет пытаться найти ближайшие ноды к уже разведанной (в этом он отличается от BFS, который исследует ноды, ближайшие к стартовой) и будет делать это на каждой последующей итерации до тех пор, пока не окажется в ноде, из которой ему некуда уйти. Тогда алгоритм отступает назад и пытается продвинуться дальше по другому пути. Алгоритм так же останавливается, когда все доступные ноды будут разведаны.

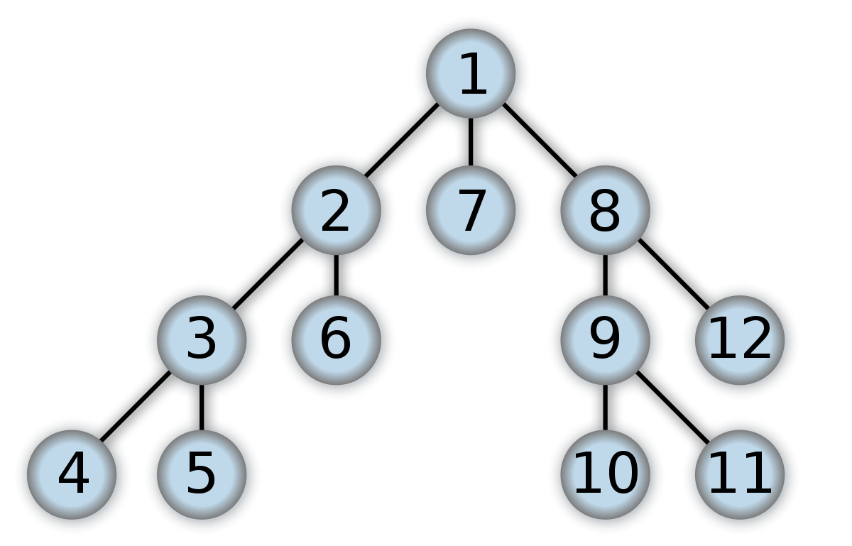
DFS реализуется на основе стека LIFO (last in, first out, «последним пришёл — первым ушёл») [6].

Рисунок 2 – Поиск в глубину

## 1.5 Жадный алгоритм

Жадный алгоритм (greedy algorithm) - это метод решения задач, при котором на каждом шаге выбирается лучшее решение на основе текущей информации, без учета будущих последствий. Такой подход позволяет быстро получить приближенное решение задачи, но не гарантирует оптимальность результата.

Принцип работы жадного алгоритма заключается в том, что на каждом шаге выбирается локально оптимальное решение, которое в итоге должно привести к глобально оптимальному решению. Жадный алгоритмы часто используются для оптимизации задач, связанных с вычислительной сложностью, таких как задачи на графах, задачи на динамическое программирование и другие.

Однако жадный алгоритм не всегда дает наилучший результат, так как он не учитывает будущие последствия своих решений. Например, если выбирать самый выгодный предмет на каждом шаге, то в итоге может получиться так, что в рюкзаке не окажется места для более ценных предметов, которые могли бы поместиться, если бы были выбраны на более поздних шагах.

# 2. Реализация алгоритма

2.1 Реализация алгоритма Дейкстры

1. Создайте пустой словарь для хранения расстояний от начальной вершины до всех остальных вершин.

2. Устанавливаем расстояние от начальной вершины до нее самой равным 0, а расстояние до всех остальных вершин равным бесконечности.

3. Добавляем начальную вершину в очередь.

4. Пока очередь не пуста, извлекаем из нее вершину с наименьшим расстоянием.

5. Для каждой соседней вершины, если расстояние от начальной вершины до данной через меньше расстояние, чем текущее расстояние до вершины, обновляем расстояние и добавляем вершину в очередь.

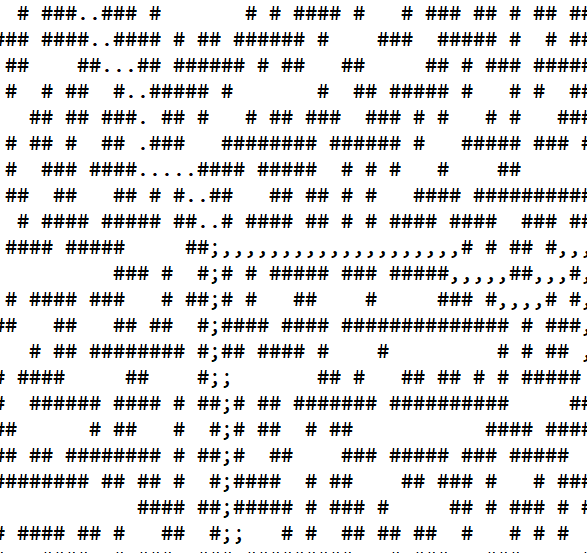
6. Повторяем шаги 4-5 для всех вершин в графе.

2.2 Реализация алгоритма A\*

Алгоритм А\* не сильно отличается от алгоритма Дейкстры, зачастую он работает более эффективнее, ведь работает направленно на конечную цель, но это может так же дать обратный результат.

1. Создайте пустой словарь для хранения рассчитанных расстояний от начальной вершины до всех остальных вершин и инициализируйте его бесконечными значениями. Установите расстояние от начальной вершины до нее самой равным 0.
2. Создайте пустой словарь для хранения родительских вершин.
3. Создайте пустое множество для хранения посещенных вершин.
4. Создайте список открытых вершин и добавьте в него начальную вершину.
5. Начните цикл, который будет выполняться до тех пор, пока все вершины не будут посещены или не будет найден путь до конечной вершины.
6. Из открытых вершин выберите вершину с наименьшей стоимостью, которая равна сумме расстояния от начальной вершины до текущей вершины и эвристической оценки расстояния от текущей вершины до конечной вершины.
7. Если текущая вершина является конечной, то путь найден. В противном случае добавьте текущую вершину в множество посещенных.
8. Обойдите все соседние вершины текущей вершины и обновите их расстояние, если оно больше, чем расстояние от начальной вершины до текущей вершины плюс вес ребра между текущей вершиной и соседней вершиной. Если расстояние было обновлено, то добавьте соседнюю вершину в список открытых вершин и установите ее родительскую вершину.
9. Повторяйте шаги 6-8, пока все вершины не будут посещены или не будет найден путь до конечной вершины.
10. Если путь был найден, то по родительским вершинам можно восстановить путь от начальной вершины до конечной. Верните список вершин, составляющих путь [3].

# 3. Пример работы

****

“#” - стена

“.” – путь от входа до ключа

“;” – места, где путь от входа до ключа и от ключа до входа пересекаются

“,” – путь от ключа до выхода

# Заключение

В результате выполненной работы была разработана программа, которая решает задачу поиска пути в лабиринте. Для этого были использованы алгоритмы Дейкстры и А\*.

Алгоритм Дейкстры основан на полном переборе возможных вариантов пути и находит кратчайший путь от одной точки до другой.

Алгоритм А\* учитывает препятствия и оценивает расстояние до цели, что позволяет находить кратчайший путь от ключа до выхода.

Таким образом, данная программа иллюстрирует работу двух различных алгоритмов поиска пути, использующих разные подходы и оценки.

# Список литературы

1. Алгоритмы: построение и анализ: учебник / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — 2-е изд. — М.: [«Вильямс»](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D0%BB%D1%8C%D1%8F%D0%BC%D1%81_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)&action=edit&redlink=1), 2006. — 1296 с. (дата обращения: 27.05.2023).
2. A\* search algorithm[Электронный ресурс]: Википедия. Свободная энциклопедия. – Режим доступа: [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm) (дата обращения: 27.05.2023).
3. Amit P. Introduction to the A\* Algorithm / P. Amit. – Текст: электронный // Red Blob Games: интернет-портал. – URL: <https://www.redblobgames.com/pathfinding/a-star/introduction.html> (дата обращения: 27.05.2023).
4. Дольников, В. Л**.** Основные алгоритмы на графах: текст лекций / В. Л. Дольников, О. П. Якимова. - Ярославль: ЯрГУ, 2011. – 80 с. – URL: <http://www.lib.uniyar.ac.ru/edocs/iuni/20110210.pdf> (дата обращения: 27.05.2023). – Текст: электронный
5. Алгоритмы поиска пути в графе // tproger.ru URL: <https://tproger.ru/articles/pathfindings/> (дата обращения: 27.05.2023).
6. Алгоритмы поиска в ширину и в глубину // konstantinklepikov.github.io URL: <https://konstantinklepikov.github.io/2020/10/09/breadth-first-and-depth-first-search-algorithms.html> (дата обращения: 27.05.2023).
7. "Greedy Algorithm", GeeksforGeeks, [Электронный ресурс] URL: <https://www.geeksforgeeks.org/greedy-algorithms/> (дата обращения: 27.05.2023).
8. "Greedy Algorithms", Brilliant, [Электронный ресурс] URL: <https://brilliant.org/wiki/greedy-algorithm/> (Дата обращения: 29.04.2023)

# Приложение 1

## Листинг программы

import heapq

# функция для чтения txt файла

def readmaze(file\_name):

with open(file\_name) as f:

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

return maze

# вход и выход

def door(maze):

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

if maze[0][X] == " ":

start = (0, X)

break

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

if maze[len(maze) - 1][X] == " ":

end = (len(maze) - 1, X)

break

return start, end

# поиск ключа

def find\_key\_position(maze):

for i, g in enumerate(maze):

if "\*" in g:

return(i, g.index("\*"))

# Поиск пути от входа до ключа с помощью алгоритма Дейкстры

def Dijkstra\_algorithm(maze, start, end):

distances = {start: 0}

queue = [(0, start, [start])]

while queue:

(cost, current, path) = heapq.heappop(queue)

if current == end:

return path

if current not in distances or cost > distances[current]:

continue

for neighbor in available\_paths(current, maze):

if neighbor not in distances or cost + 1 < distances[neighbor]:

distances[neighbor] = cost + 1

heapq.heappush(queue, (cost + 1, neighbor, path + [neighbor]))

return None

# функция для поиска доступных путей из текущей точки

def available\_paths(current, maze):

neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]

paths = []

for neighbor in neighbors:

x = current[0] + neighbor[0]

y = current[1] + neighbor[1]

if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] != "#":

paths.append((x, y))

return paths

# нахождения пути от ключа до выхода с помощью алгоритма А\*

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

heap = [(0, start)]

heapq.heapify(heap)

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)

return path[::-1]

visited.add(current)

for neighbor in available\_paths(current, maze):

if neighbor == start or neighbor == key:

continue

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] = g\_score[neighbor] + heuristic(neighbor, end)

if neighbor not in visited:

heapq.heappush(heap, (f\_score[neighbor], neighbor))

return []

# функция для определения эвристической оценки расстояния между двумя точками

def heuristic(current, end):

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

# создание нового лабиринта с отмеченными путями

def new\_maze(maze, path, marker):

for i, j in path:

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

if marker == "." and (i, j) in path2:

maze[i][j] = ";"

else:

maze[i][j] = marker

return maze

# функция для записи обновленного лабиринта в txt файл

def new\_file(maze, file\_name):

with open(file\_name, "w") as f:

for g in maze:

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

maze = readmaze("maze-for-u.txt")

start, end = door(maze)

key = find\_key\_position(maze)

path1 = Dijkstra\_algorithm(maze, start, key)

path2 = A\_star(maze, key, end)

maze = new\_maze(maze, path1, ".")

maze = new\_maze(maze, path2, ",")

new\_file(maze, "maze-for-me-done.txt")