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 Редькин М.Д.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc136392347)

[Задачи 3](#_Toc136392348)

[1.Теоретическая часть 4](#_Toc136392349)

[2. Реализация алгоритмов 7](#_Toc136392350)

[Пример работы 10](#_Toc136392351)

[Заключение 11](#_Toc136392352)

[Список литературы 12](#_Toc136392353)

[Приложение 1 13](#_Toc136392354)

[Листинг программы 13](#_Toc136392355)

# Введение

Алгоритмы обхода графа являются важнейшей задачей в программировании, так как многие проблемы, связанные с обработкой данных, могут быть представлены в виде графа. Например, поиск кратчайшего пути между двумя точками, определение связности компонентов и многие другие задачи могут быть эффективно решены с помощью алгоритмов обхода графа. Один из таких алгоритмов - алгоритм Дейкстры, который позволяет находить кратчайший путь во взвешенном графе с неотрицательными весами ребер [1].

Кроме того, для решения задачи поиска маршрута в лабиринте может быть использован алгоритм A\*. Он является модификацией алгоритма Дейкстры и позволяет учитывать не только стоимость пути от начальной вершины до текущей, но и эвристическую оценку расстояния до конечной вершины [2]

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

# Задачи

• Изучить алгоритмы поиска маршрута в графе;

• Определить особенности реализации, необходимые для конкретной задачи поиска маршрута;

• Подготовить исходные данные: лабиринт, координаты точек для посещения при обходе;

• Написать код для реализации алгоритмов с заданными параметрами;

• Сохранить результаты обходов лабиринта и полученные маршруты в файл.

# 1.Теоретическая часть

**Алгоритм Дейкстры**

**Алгоритм Дейкстры** назван в честь голландского ученого Эдсгера Дейкстры (Edsger Dijkstra). Алгоритм был предложен в 1959 году для нахождения кратчайших путей от одной вершины до всех остальных в ориентированном взвешенном графе, при условии, что все ребра в графе имеют неотрицательные веса.

Алгоритм Дейкстры работает на ориентированных (с некоторыми дополнениями и на неориентированных) графах, и призван искать кратчайшие пути между заданной вершиной и всеми остальными вершинами в графе.  
 Как правило, граф обозначают как набор вершин и рёбер G = (V, E) где число рёбер может быть задано m, а вершин числом n

Для каждого ребра в графе задан неотрицательный вес l, а также вершина, из которой осуществляется поиск оптимальных путей.

Алгоритм Дейкстры может найти кратчайший путь между вершинами s и t в графе, только если существует хотя бы один путь между этими вершинами. Если это условие не выполняется, то алгоритм отработает корректно, вернув значение "бесконечность" для пары несвязанных вершин.

Условие неотрицательности весов рёбер крайне важно и от него нельзя просто избавиться. Не получится свести задачу к решаемой алгоритмом Дейкстры, прибавив наибольший по модулю вес ко всем рёбрам. Это может изменить оптимальный маршрут

**Кем используется алгоритм Дейксты**

* Математики и другие ученые, которые пользуются графами как абстрактными единицами. Задача поиска маршрута в науке может быть и чисто фундаментальной, и прикладной.
* Дата-сайентисты. В этой области много математики, в том числе активно используется теория графов.
* Сетевые инженеры, так как алгоритм Дейкстры лежит в основе работы нескольких протоколов маршрутизации. Сама по себе компьютерная сеть представляет собой граф, поэтому специалисты по сетям должны знать, что это такое.

**Для чего нужен алгоритм Дейкстры**

Основная задача — поиск кратчайшего пути по схеме, где множество точек соединено между собой отрезками. В виде такой схемы можно представить многие объекты реального мира, поэтому практических примеров использования алгоритма много:

* автоматическое построение маршрута на онлайн-карте;
* поиск системой бронирования наиболее быстрых или дешевых билетов, в том числе с возможными пересадками;
* моделирование движения робота, который перемещается по местности;
* разработка поведения неигровых персонажей, создание игрового ИИ в геймдеве;
* автоматическая обработка транспортных потоков;
* маршрутизация движения данных в компьютерной сети;
* расчет движения тока по электрическим цепям.

**A Star**

Алгоритм «A Star» — это один из проверенных опытом алгоритмов, который используют для того, чтобы найти кратчайший путь между 2 вершинами графа, у которых положительный вес ребер. Данный алгоритм был описан еще в 1968 году П. Хартом, Н. Нильсоном и Б. Рафаэлем.

Алгоритм «A Star» всегда находит кратчайший путь, потому что применяет в своих расчетах вспомогательную функцию, которая всегда направляет поиск в нужном направлении, сокращая его длительность. Такая вспомогательная функция называется «эвристика», поэтому алгоритм «A Star» относят к категории «эвристических алгоритмов поиска».

**Алгоритм A\***

Алгоритм A Star характеризуется тремя важными свойствами:

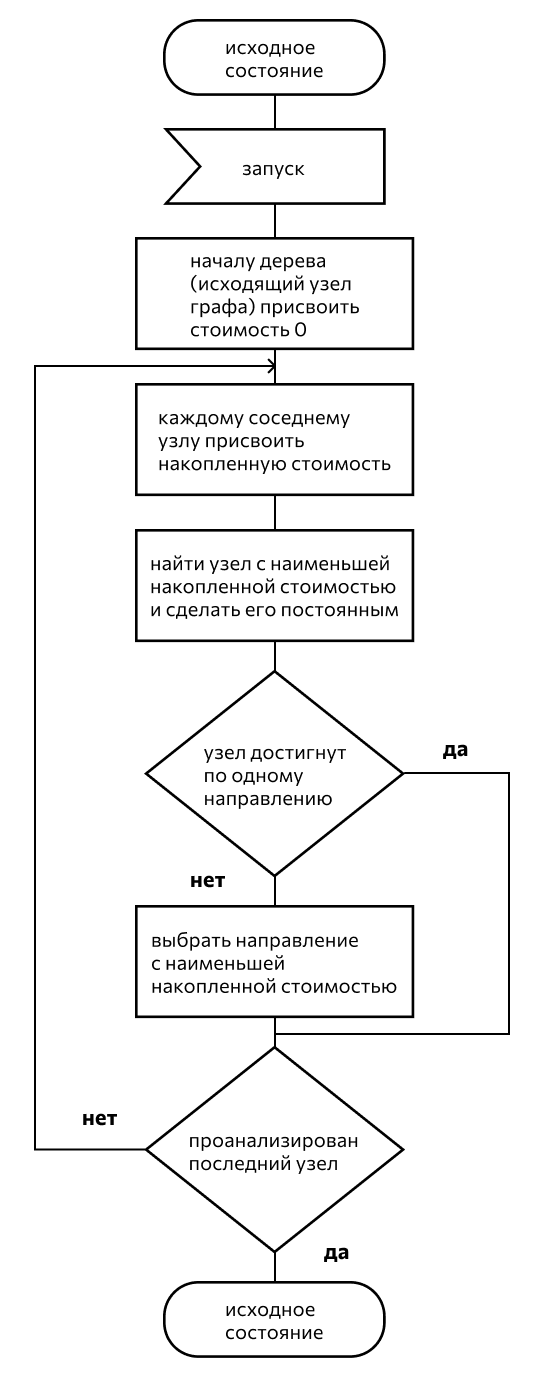
* оптимальностью — это означает, что алгоритм гарантирует получение лучшего из возможных результатов;
* полнотой — это означает, что алгоритм A\* всегда найдет решение, если оно существует;
* эффективностью — на сегодняшний день нет других алгоритмов, которые смогут найти кратчайший путь быстрее, чем A\*, применяя эвристическую функцию.

Алгоритм поиска A\* несет в себе следующую идею: изначально он всегда посещает вершины, которые, скорее всего, ведут по кратчайшему пути к цели. Такие вершины он определяет по формуле: F(x) = G(x) + H(x) , где:

* F(x) — это функция для вершины; чем меньше функция, тем «ближе» вершина стоит в очереди для посещения; данная функция оценивает минимальную стоимость перехода от вершины к вершине;
* G(x) — это стоимость пути от первоначальной вершины и до любой другой;
* H(x) — это эвристический показатель стоимости пути от вершины «х» и до конечной вершины.

Вычисляя F(x), за «х» берется соседняя вершина с уже посещенной. Поэтому получается, что алгоритм вычисляет стоимость всего пути у всех соседних вершин и останавливает выбор на той, у которой стоимость минимальна.

# 2. Реализация алгоритмов

**Как работает алгоритм Дейкстры**

Алгоритм Дейкстры пошаговый. Сначала выбирается точка, от которой будут отсчитываться пути. Затем алгоритм поочередно ищет самые короткие маршруты из исходной точки в другие. Вершины, где он уже побывал, отмечает посещенными. Алгоритм использует посещенные вершины, когда рассчитывает пути для непосещенных.

**Как работает алгоритм A\***

Посетив одну конкретную вершину, алгоритм A\* перед переходом к следующей исследует все соседние вершины. Все вершины алгоритм разделяет на 3 категории:

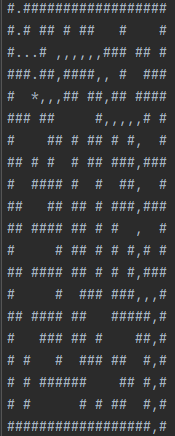
* Неизвестные вершины. Это те, которые не были еще посещены и пока что даже не найдены. Получается, что и путь к ним пока остается загадкой. Таким образом, изначально все вершины, кроме стартовой, будут в этой категории.
* Известные вершины. Это те вершины, о которых уже известно алгоритму и уже даже известен путь к ним. Такие вершины сохраняются в «списке алгоритма» и становятся в очередь для их посещения и исследования. Из этого списка исследуются те вершины, которые считаются наиболее перспективными.
* Исследованные вершины. В эту категорию попадают те вершины, которые уже были посещены алгоритмом A\*. К этим вершинам известен самый короткий путь, поэтому они попадают в «закрытый список» — этот список нужен для того, чтобы исключить многократное исследование одних и тех же вершин.

Когда одна из вершин становится полностью исследованной, она попадает в категорию «исследованные вершины», а все ее соседи попадают в категорию «известные вершины» и становятся годными для исследования. На каждой уже исследованной вершине устанавливается указатель до той уже исследованной вершины, к которой у нее будет кратчайший путь.

Алгоритм A\* завершает свою работу только в том случае, если конечная вершина переносится в категорию «исследованные вершины». В этом случае уже будет весь список исследованных вершин, а на каждой из них будет стоять указатель с кратчайшим путем. Поэтому несложно будет по указателям отследить кратчайший путь от конечной вершины до начальной.

Алгоритм A\* находит кратчайший путь между вершинами, основываясь на стоимости и «весе» ребер. Поэтому путь, который находит A\*, можно по праву назвать «самым быстрым» или «самым простым». По этой причине алгоритм A\* очень часто применяется как раз для планирования кратчайших путей, однако его также часто применяют в играх.

# Пример работы

****

# Заключение

В рамках работы были применены алгоритмы Дейкстры и А\* для поиска кратчайшего маршрута в лабиринте. Были изучены особенности реализации данных алгоритмов и подготовлены исходные данные для обхода лабиринта. Результаты обходов были сохранены в файлы. В итоге, цель работы была достигнута, а задачи выполнены успешно. Разработанная программа на языке программирования Python может быть использована для решения задач по поиску выхода из лабиринта и других задач поиска маршрута в графе.

# Список литературы

1. «Алгоритм Дейкстры» [Электронный ресурс]. URL: https://habr.com/ru/companies/otus/articles/599621/ (Дата обращения: 25.05.2023).
2. «Поисковый алгоритм A Star: что это и как эффективно его использовать?» [Электронный ресурс]. URL: <https://codernet.ru/articles/web/poiskovyij_algoritm_a_star_chto_eto_i_kak_effektivno_ego_ispolzovat/> (Дата обращения: 25.05.2023).

# Приложение 1

## Листинг программы

import heapq  
from typing import List, Tuple  
  
def read\_maze(file\_name):  
 with open(file\_name, "r") as f:  
 lines = f.read().splitlines()  
 maze = [list(line) for line in lines]  
 return maze  
  
def start\_end\_points(maze):  
 start = None  
 end = None  
 for i in range(len(maze[0])):  
 if maze[0][i] == " ":  
 start = (0, i)  
 if maze[len(maze)-1][i] == " ":  
 end = ((len(maze)-1), i)  
 return start, end  
  
def key\_points(maze):  
 key = None  
 for i in range(len(maze)):  
 for j in range(len(maze[0])):  
 if maze[i][j] == "\*":  
 key = (i, j)  
 return key  
  
def Dijkstra(start: Tuple[int, int], end: Tuple[int, int], maze: List[List[int]]) -> List[Tuple[int, int]]:  
 m, n = len(maze), len(maze[0])  
 visited = {(i, j): False for i in range(m) for j in range(n)}  
 distance = {(i, j): float("inf") for i in range(m) for j in range(n)}  
 previous = {(i, j): None for i in range(m) for j in range(n)}  
 distance[start] = 0  
 queue = {start}  
 while len(queue) > 0:  
 u = min(queue, key=lambda x: distance[x])  
 queue.remove(u)  
 visited[u] = True  
 if u == end:  
 path = []  
 while previous[u]:  
 path.append(u)  
 u = previous[u]  
 path.append(start)  
 path.reverse()  
 return path  
 for v in [(u[0]-1, u[1]), (u[0]+1, u[1]), (u[0], u[1]-1), (u[0], u[1]+1)]:  
 if (0 <= v[0] < m) and (0 <= v[1] < n) and not visited[v] and maze[v[0]][v[1]] != "#":  
 alt = distance[u] + 1  
 if alt < distance[v]:  
 distance[v] = alt  
 previous[v] = u  
 queue.add(v)  
 return []  
  
def Astar(start: Tuple[int, int], end: Tuple[int, int], maze: List[List[str]]) -> List[Tuple[int, int]]:  
  
 def Heurit(a: Tuple[int, int], b: Tuple[int, int]) -> float:  
 return abs(b[0] - a[0]) + abs(b[1] - a[1])  
  
 open\_set = [(0, start)]  
 closed\_set = set()  
  
 expenses = {start: 0}  
 came\_from = {}  
  
 while open\_set:  
 Now = heapq.heappop(open\_set)[1]  
 if Now == end:  
 path = []  
 while Now in came\_from:  
 path.append(Now)  
 Now = came\_from[Now]  
 path.append(start)  
 path.reverse()  
 return path  
 closed\_set.add(Now)  
 for row, col in [(Now[0] - 1, Now[1]), (Now[0] + 1, Now[1]), (Now[0], Now[1] - 1), (Now[0], Now[1] + 1)]:  
 if 0 <= row < len(maze) and 0 <= col < len(maze[0]) and maze[row][col] != "#":  
 estimated\_costs = expenses[Now] + 1  
 if (row, col) in closed\_set and estimated\_costs >= expenses.get((row, col), float("inf")):  
 continue  
 if estimated\_costs < expenses.get((row, col), float("inf")):  
 expenses[(row, col)] = estimated\_costs  
 came\_from[(row, col)] = Now  
 heapq.heappush(open\_set, (estimated\_costs + Heurit((row, col), end), (row, col)))  
 return []  
  
def new\_maze(maze, path, mark):  
 for cord in path:  
 x, y = cord  
 maze[x][y] = mark  
 return maze  
  
def new\_file(maze, file\_name):  
 with open(file\_name, "w") as file:  
 for row in maze:  
 for elem in row:  
 file.write(str(elem))  
 file.write("\n")  
  
maze = read\_maze("maze-for-u.txt")  
start, end = start\_end\_points(maze)  
key = key\_points(maze)  
  
path1 = Dijkstra(start, key, maze)  
path2 = Astar(key, end, maze)  
  
maze = new\_maze(maze, path1, ".")  
maze = new\_maze(maze, path2, ",")  
x, y = key  
maze[x][y] = "\*"  
  
new\_file(maze, "maze-for-me-done.txt")