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 Шохинов Н.С.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc1)

[Задачи 3](#_Toc2)

[1.Теоретическая часть 4](#_Toc3)

[2. Реализация алгоритма 5](#_Toc4)

[Пример работы 6](#_Toc5)

[Заключение 7](#_Toc6)

[Список литературы 8](#_Toc7)

[Приложение 1 9](#_Toc8)

[Листинг программы 9](#_Toc9)

# 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 [1]. Одним из таких является алгоритм Дейкстры [2].

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

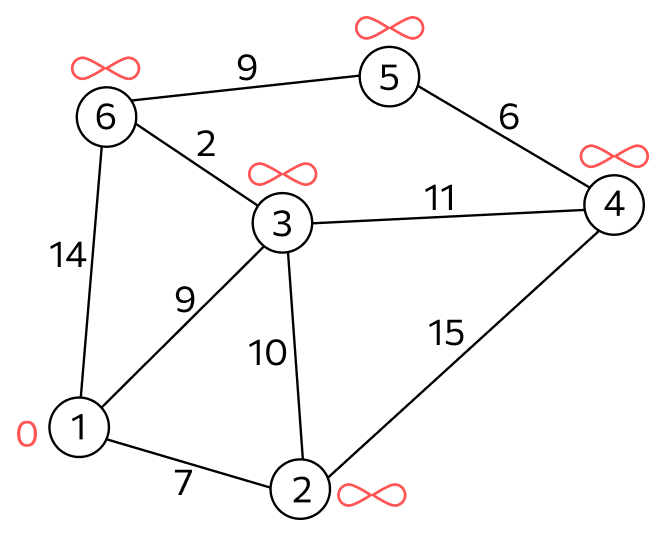
# Задачи

* 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;
* Реализовать алгоритмы с заданными параметрами;
* Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

# 1.Теоретическая часть

Алгоритм Дейкстры

Алгоритм Дейкстры — это метод, который находит кратчайший путь от одной вершины графа к другой. Граф — структура из точек-вершин, соединенных ребрами-отрезками. Его можно представить как схему дорог или как компьютерную сеть. Ребра — это связи, по ним можно двигаться от одной вершины к другой.



Графы используют для моделирования реальных объектов, а алгоритмы поиска пути — при их изучении, а также решении практических задач. Алгоритм Дейкстры работает для графов, у которых нет ребер с отрицательным весом, т.е. таких, при прохождении через которые длина пути как бы уменьшается.

В отличие от похожих методов, алгоритм Дейкстры ищет оптимальный маршрут от одной заданной вершины ко всем остальным. Попутно он высчитывает длину пути — суммарный вес ребер, по которым проходит при этом маршруте.

Кто пользуется алгоритмом Дейкстры

* 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.

Зачем 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.

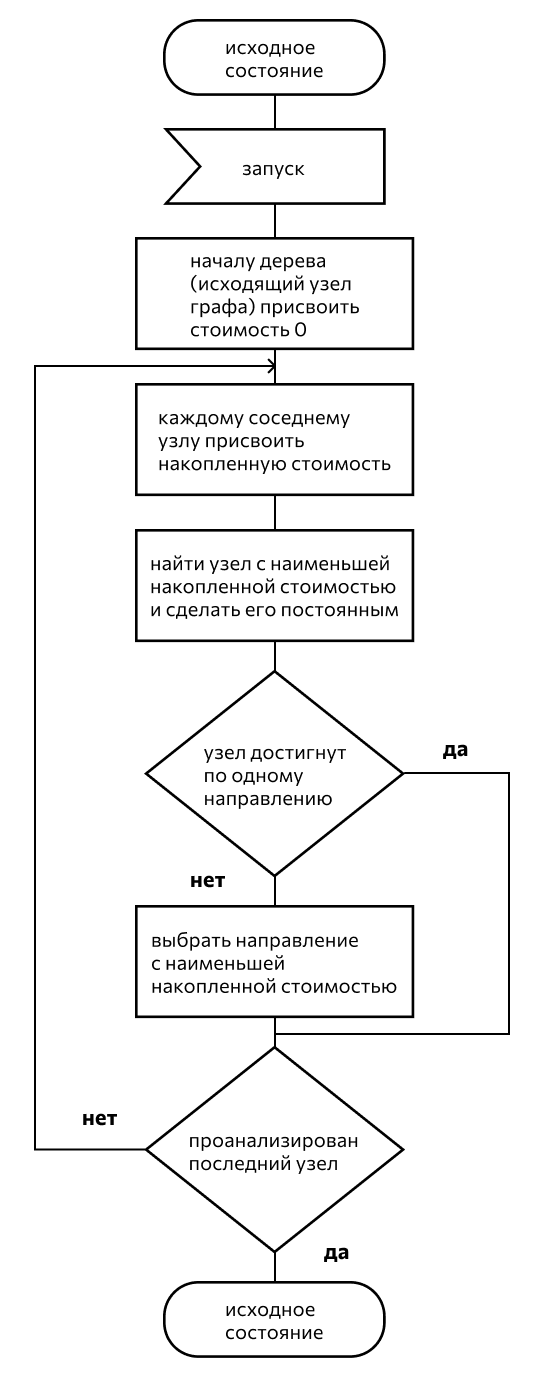
Алгоритм поиска A\* несет в себе следующую идею: изначально он всегда посещает вершины, которые, скорее всего, ведут по кратчайшему пути к цели. Такие вершины он определяет по формуле: F(x) = G(x) + H(x) , где:

* 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 алгоритма

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:

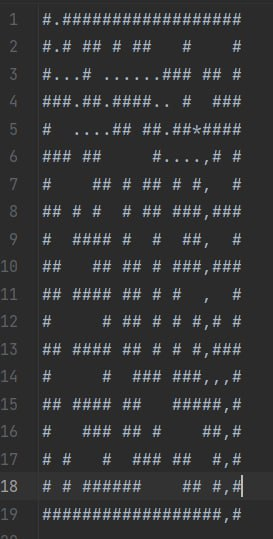
* Неизвестные вершины. Это те, которые не были еще посещены и пока что даже не найдены. Получается, что и путь к ним пока остается загадкой. Таким образом, изначально все вершины, кроме стартовой, будут в этой категории.
* Известные вершины. Это те вершины, о которых уже известно алгоритму и уже даже 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 ходе работы были реализованы алгоритмы обхода графа Дейкстры и А\* для задачи поиска маршрута в лабиринте. Были изучены особенности их реализации и подготовлены исходные данные для обходов лабиринта. Результаты обходов сохранены в файлы. В целом, цель работы была достигнута, а задачи выполнены успешно. Реализованные алгоритмы могут быть использованы в различных задачах поиска маршрута в графе.

В результате была разработана программа, которая способна решать задачи по поиску выхода из лабиринта на языке программирования python.

# Список литературы

1. Шагин А. «Графы: основы теории, алгоритмы поиска» [Электронный ресурс]. URL: <https://medium.com/nuances-of-programming/графы-основы-теории-алгоритмы-поиска-b93672f59747> (Дата обращения: 26.04.2023).
2. «Алгоритм Дейкстры» [Электронный ресурс]. URL: <https://blog.skillfactory.ru/glossary/algoritm-dejkstry/> (Дата обращения: 26.04.2023).
3. «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: 26.04.2023).

# Приложение 1

## Листинг программы

import heapq

from typing import List, Tuple

def read\_maze(filename):

    with open(filename, "r") as f:

        lines = f.read().splitlines()

    maze = [list(line) for line in lines]

    return maze

def find\_points(maze):

    start = None

    end = None

    for i in range(len(maze[0])):

        if maze[0][i] == " ":

            start = (0, i)

        if maze[i][0] == " ":

            start = (i, 0)

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

            end = ((len(maze)-1), i)

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

            end = (i, (len(maze)-1))

    return start, end

def find\_key(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)}

    prev = {(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 prev[u]:

                path.append(u)

                u = prev[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

                    prev[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 heuristic(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()

    cost = {start: 0}

    came\_from = {}

    while open\_set:

        current = heapq.heappop(open\_set)[1]

        if current == end:

            path = []

            while current in came\_from:

                path.append(current)

                current = came\_from[current]

            path.append(start)

            path.reverse()

            return path

        closed\_set.add(current)

        for row, col in [(current[0] - 1, current[1]), (current[0] + 1, current[1]), (current[0], current[1] - 1), (current[0], current[1] + 1)]:

            if 0 <= row < len(maze) and 0 <= col < len(maze[0]) and maze[row][col] != "#":

                tentative\_cost = cost[current] + 1

                if (row, col) in closed\_set and tentative\_cost >= cost.get((row, col), float("inf")):

                    continue

                if tentative\_cost < cost.get((row, col), float("inf")):

                    cost[(row, col)] = tentative\_cost

                    came\_from[(row, col)] = current

                    heapq.heappush(open\_set, (tentative\_cost + heuristic((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, filename):

    with open(filename, "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") # test.txt

start, end = find\_points(maze)

key = find\_key(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") # test\_new2.txt