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 Семенов А.А.

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 поиска пути и структурное программирование являются важными инструментами в разработке программного обеспечения для решения задач, связанных с поиском оптимальных путей в различных ситуациях, например, в навигации или транспортной логистике. В данной курсовой работе рассматриваются основные алгоритмы поиска пути и их реализация с помощью структурного программирования.

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

# Задачи

* Изучить алгоритмы Дейкстры и А\*;
* Выделить особенности реализации, необходимые в конкретной задаче поиска маршрута;
* Подготовить исходные данные: лабиринт, координаты точек для посещения при обходе;
* Реализовать алгоритмы с заданными параметрами;
* Создать новый файл для хранения лабиринта с реализованными алгоритмами и отмеченным путём.0--------

# 1.Теоретическая часть

**Алгоритмы поиска пути**

Алгоритм поиска пути — это математический алгоритм, который позволяет найти кратчайший или оптимальный путь между двумя точками на графе, сетке или карте. Он широко используется в различных областях, таких как логистика, робототехника, компьютерные игры, маршрутизация сетей, маршрутизация транспорта и другие.

По своей сути алгоритм поиска пути ищет на графе, начиная с одной (стартовой) точки и исследуя смежные узлы до тех пор, пока не будет достигнут узел назначения (конечный узел). Кроме того, в алгоритмах поиска пути в большинстве случаев заложена также цель найти самый короткий путь. Некоторые методы поиска на графе, такие как поиск в ширину, могут найти путь, если дано достаточно времени. Другие методы, которые «исследуют» граф, могут достичь точки назначения намного быстрее.

Основными задачами алгоритмов поиска пути являются:

* Нахождение кратчайшего пути между двумя точками.
* Оптимизация пути, учитывая различные ограничения и препятствия на пути.
* Построение оптимального маршрута для доставки грузов, обслуживания городов и других задач.
* Определение оптимального пути для навигации роботов и других устройств.

Существует множество алгоритмов поиска пути, каждый из которых имеет свои преимущества и недостатки в зависимости от конкретной задачи. В данной курсовой работе будут рассмотрены алгоритм Дейкстры и алгоритм поиска A\*.

**Алгоритм Дейкстры**

Алгоритм Дейкстры – это алгоритм поиска кратчайшего пути в графе с неотрицательными весами ребер. Он был разработан в 1956 году голландским ученым Эдсгером Дейкстрой и является одним из наиболее известных и используемых алгоритмов в современной математике и информатике.

Основным принципом алгоритма Дейкстры является постепенное нахождение кратчайших путей от начальной вершины до всех остальных вершин графа. В процессе работы алгоритма вершины графа разбиваются на две группы – множество посещенных вершин и множество не посещенных вершин, начиная с начальной вершины.

В некоторых областях, в частности в искусственном интеллекте, алгоритм Дейкстры или его вариант известен как поиск по равномерной стоимости и сформулирован как пример более общей идеи поиска по принципу наилучшего первого.

Алгоритм Дейкстры может применяться в различных областях, включая телекоммуникационные сети, GPS-навигацию, планирование маршрутов, решение задач маршрутизации и многое другое.

Одной из основных областей применения алгоритма Дейкстры являются телекоммуникационные сети. Например, в VoIP-системах он используется для поиска кратчайшего пути между двумя телефонами, а в сетях связи – для оптимизации маршрутов передачи данных.

**Алгоритм поиска А\***

Алгоритм поиска A\* является одним из наиболее эффективных алгоритмов поиска кратчайшего пути в графах с весами на ребрах. Он объединяет в себе идеи алгоритмов Дейкстры и Левита, позволяя учитывать, как стоимость достижения вершины из начальной, так и эвристическую оценку расстояния от вершины до конечной точки. Эвристическая оценка позволяет сократить количество рассматриваемых вершин на основе некоторых знаний о структуре графа.

Алгоритм работает на основе понятия "остаточная стоимость" – это сумма стоимости достижения вершины из начальной точки и эвристической оценки стоимости до конечной точки. Остаточную стоимость определяется для каждой вершины при ее рассмотрении алгоритмом.

Процесс работы алгоритма начинается с установки начальной точки. Затем алгоритм ищет вершину с наименьшей остаточной стоимостью и помечает ее как текущую вершину. Далее производятся раскрытие текущей вершины и оценка всех ее соседей путем вычисления их остаточной стоимости. Если она меньше, чем уже сохраненное значение остаточной стоимости для соседа, то происходит обновление значения. После этого текущей вершиной становится та, у которой наименьшее значение остаточной стоимости, и процесс повторяется, пока не будет найден кратчайший путь до конечной точки.

# 2. Реализация алгоритмов

Алгоритм Дейкстры включает в себя следующие этапы:

1. Инициализация. Создать массив, в котором будут храниться веса вершин. Изначально, вес каждой вершины устанавливается равным бесконечности, за исключением начальной точки, вес которой устанавливается равным 0.
2. Отметить начальную вершину как активную.
3. Найти все соседние вершины активной вершины.
4. Обновить вес каждой соседней вершины путем сложения веса активной вершины и веса ребра, соединяющего активную вершину и соседнюю вершину.
5. Пометить активную вершину как просмотренную.
6. Выбрать следующую не просмотренную вершину с наименьшим весом.
7. Повторять шаги 3-6 для каждой вершины графа.
8. Результатом работы алгоритма является массив весов вершин, который показывает кратчайшее расстояние от начальной точки до каждой вершины графа.

Алгоритм А\* включает в себя следующие этапы:

1. Инициализируем открытый список, закрытый список, начальную вершину и целевую вершину.
2. Добавляем начальную вершину в открытый список.
3. Пока открытый список не пуст, выбираем вершину с наименьшей оценкой f из открытого списка.

* если выбранная вершина - целевая, находим оптимальный путь и возвращаем его.
* перемещаем выбранную вершину в закрытый список и перебираем соседние вершины.

1. Для каждой соседней вершины:

* если вершина уже в закрытом списке, игнорируем ее.
* если вершина не в открытом списке, добавляем ее в открытый список.
* оцениваем расстояние от начальной вершины до этой вершины g.
* оцениваем расстояние от этой вершины до целевой вершины h.
* вычисляем оценку f = g + h.
* если новое значение f меньше, чем старое, обновляем значения g и f вершины и устанавливаем ее родителем выбранную вершину.

# Пример работы

Для примера работы был взят небольшой лабиринт. “#” – стены лабиринта. Точками проложен маршрут, с помощью алгоритма Дейкстры, от входа в лабиринт до ключа, а путь от ключа до выхода из лабиринта состоит из запятых. Результат можно посмотреть на рисунке 1.



Рисунок 1

# Заключение

В заключении хочется отметить, что изучение алгоритмов Дейкстры и A\* для нахождения кратчайшего пути в графах является важной задачей в информатике и программировании.

Алгоритм Дейкстры работает с положительными весами ребер и находит кратчайшие пути от начальной вершины до всех остальных. Он является оптимальным алгоритмом для решения такой задачи, но может работать недостаточно быстро на больших графах.

A\*, в свою очередь, использует эвристику для быстрого нахождения кратчайшего пути между двумя вершинами в графе. Он является более эффективным, чем алгоритм Дейкстры.

В целом, оба алгоритма являются полезными в программировании и находят применение в широком спектре задач. Однако, при выборе конкретного алгоритма необходимо учитывать особенности задачи и графа, на котором он будет использоваться.

Результатом данной работы является программа, которая способна находить самый оптимальный путь, используя алгоритмы А\* и Дейкстры.

# Список литературы

1. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы [Электронный ресурс]. URL: <https://djvu.online/file/9vPpNdLjbxy8q> (дата обращения 19.05.2023).
2. Стивенс Род. Алгоритмы. Теория и практическое применения [Электронный ресурс] / Род Стивенс – Москва, 2016 – 544 с. URL: <https://library.samdu.uz/files/6d05d440dc21241128f485ae90f2422b_Essential%20%20Algorithms.pdf> (дата обращения: 19.05.2023
3. Евкова А. Понятие и основные свойства алгоритмов [Электронный ресурс]. URL: <https://www.evkova.org/kursovye-raboty/ponyatie-i-osnovnyie-svojstva-algoritmov> (дата обращения: 20.05.2023)
4. М. Нидхем, Холдер Э. Графовые алгоритмы. Практическая реализация на платформах ApacheSpark и Neo4j. [Электронный ресурс]/ пер. с англ. В. С. Яценкова – 2020. – 258 с. URL: <https://ugolok.vercel.app/books/algoritms/grafovye_algoritmy.pdf> (дата обращения: 21.05.2023)
5. Иванов М. Алгоритм Дейкстры [Электронный ресурс]. URL: <https://ru.algorithmica.org/cs/shortest-paths/dijkstra/> (дата обращения 22.05.2023).
6. Поддержка аннотации типов в Python [Электронный ресурс]. URL: <https://docs-python.ru/standart-library/modul-typing-python/> (дата обращения 22.05.2023)
7. Стоу Б. Алгоритмы поиска пути [Электронный ресурс]/ пер. с англ. М. Каменский – 2000. URL: <http://pmg.org.ru/ai/stout.htm> (дата обращения: 25.05.2023).

# Приложение 1

## Листинг программы

## import heapq

## from typing import List, Tuple

## def read\_maze(file):

## with open(file, 'r') as f:

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

## return maze

## # Функция нахождения точки входа

## def start\_pos(maze):

## for Y in range(len(maze[0])):

## if maze[0][Y] == " ":

## start = (0, Y)

## break

## return start

## # Функция нахождения точки выхода

## def end\_pos(maze):

## for Y in range(len(maze[0])):

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

## end = (len(maze) - 1, Y)

## break

## return end

## # Функция нахождения ключа

## def key\_pos(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

## #Алгоритм Дейкстры

## 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 []

## #A\*

## def astar(start: Tuple[int, int], end: Tuple[int, int], maze: List[List[str]]) -> List[Tuple[int, int]]:

## 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 []

## #Эвристическая стоимость

## def heuristic(a: Tuple[int, int], b: Tuple[int, int]) -> float:

## return abs(b[0] - a[0]) + abs(b[1] - a[1])

## 

## def updated\_maze(maze, path, mark):

## for pos in path:

## x, y = pos

## 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")

## def main():

## maze = read\_maze("test\_maze.txt")

## start = start\_pos(maze)

## end = end\_pos(maze)

## key = key\_pos(maze)

## go\_to\_key = dijkstra(start, key, maze)

## go\_to\_exit = astar(key, end, maze)

## maze = updated\_maze(maze, go\_to\_key, ".")

## maze = updated\_maze(maze, go\_to\_exit, ",")

## x, y = key

## maze[x][y] = "\*"

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

## print("Лабиринт пройден. Результат можно посмотреть в файле 'maze-for-me-done.txt'.")

## main()