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-33 Егоров К.А.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc133872358)

[Задачи 3](#_Toc133872359)

[1.Теоретическая часть 4](#_Toc133872360)

[а) Поиск в ширину 4](#_Toc133872361)

[б)Алгоритм А\* 5](#_Toc133872362)

[2. Реализация алгоритма 6](#_Toc133872363)

[a)Поиск в ширину 6](#_Toc133872364)

[б)Алгоритм А\* 6](#_Toc133872365)

[Пример работы 8](#_Toc133872366)

[a)Поиск в ширину 8](#_Toc133872367)

[б)Алгоритм А\* 8](#_Toc133872368)

[Заключение 9](#_Toc133872369)

[Список литературы 10](#_Toc133872370)

[Приложение 1 11](#_Toc133872371)

[Листинг программы 11](#_Toc133872372)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании, так как посредством графов можно описывать разнообразные реальные явления: организацию транспортных систем, сети передачи данных, человеческих взаимоотношений, структуру гена или молекулы. Возможность формального моделирования такого множества разных реальных структур позволяет программисту решать широкий круг прикладных задач[1]. Одними из этих алгоритмов являются поиск в ширину и А\*.

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

# Задачи

* Изучить алгоритмы построения маршрута в графе;
* 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

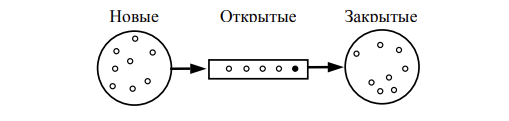
## а) Поиск в ширину

Работа всякого алгоритма обхода состоит в последовательном посещении вершин и исследовании ребер. Какие именно действия выполняются при посещении вершины и исследовании ребра – это зависит от конкретной задачи, для решения которой производится обход. В любом случае, однако, факт посещения вершины запоминается, так что с момента посещения и до конца работы алгоритма она считается посещенной. Вершину, которая еще не посещена, будем называть новой. В результате посещения вершина становится открытой и остается такой, пока не будут исследованы все инцидентные ей ребра. После этого она превращается в закрытую. [2]

Идея поиска в ширину состоит в том, чтобы посещать вершины в порядке их удаленности от некоторой заранее выбранной или указанной стартовой вершины a. Иначе говоря, сначала посещается сама вершина a, затем все вершины, смежные с a, то есть находящиеся от нее на расстоянии 1, затем вершины, находящиеся от a на расстоянии 2, и т.д. [2]

Рассмотрим алгоритм поиска в ширину с заданной стартовой вершиной a. Вначале все вершины помечаются как новые. Первой посещается вершина a, она становится единственной открытой вершиной. В дальнейшем каждый очередной шаг начинается с выбора некоторой открытой вершины x. Эта вершина становится активной. Далее исследуются ребра, инцидентные активной вершине. Если такое ребро соединяет вершину x с новой вершиной y, то вершина y посещается и превращается в открытую. Когда все ребра, инцидентные активной вершине, исследованы, она перестает быть активной и становится закрытой. После этого выбирается новая активная вершина, и описанные действия повторяются. Процесс заканчивается, когда множество открытых вершин становится пустым. [2]

Основная особенность поиска в ширину, отличающая его от других способов обхода графов, состоит в том, что в качестве активной вершины выбирается та из открытых, которая была посещена раньше других. Именно этим обеспечивается главное свойство поиска в ширину: чем ближе вершина к старту, тем раньше она будет посещена. Для реализации такого правила выбора активной вершины удобно использовать для хранения множества открытых вершин очередь – когда новая вершина становится открытой, она добавляется в конец очереди, а активная выбирается в ее начале. Схематически процесс изменения статуса вершин изображен на рисунке черным кружком показана активная вершина: [2]



## б)Алгоритм А\*

Алгоритм A\* - это эвристический алгоритм поиска кратчайшего пути, который используется во многих областях искусственного интеллекта и робототехники.

Алгоритм A\* работает с графом, состоящим из узлов и ребер, где узлы представляют собой состояния, а ребра - переходы между состояниями. A\* ищет кратчайший путь от начального узла к конечному узлу, используя эвристику для ускорения поиска.

В основе алгоритма A\* лежит комбинация двух функций - функции стоимости пути от начального узла до текущего узла (g-функция) и эвристической функции оценки стоимости пути от текущего узла до конечного узла (h-функция). Сумма g- и h-функций определяет оценочную стоимость пути от начального узла до конечного узла через текущий узел.

A\* использует информацию об оценочной стоимости пути, которая является взвешенной суммой фактической стоимости пути от начала до данной вершины и эвристической оценки стоимости пути от данной вершины до цели. Интуитивно, эвристическая функция оценки должна быть меньше или равна фактической стоимости пути. [3]

A\* строит дерево поиска, добавляя в него новые узлы из открытого списка, который содержит узлы, которые еще не были проверены. На каждой итерации алгоритм выбирает узел из открытого списка с минимальной оценочной стоимостью пути от начального узла, добавляет его дочерние узлы в открытый список и удаляет выбранный узел из открытого списка.

# 2. Реализация алгоритма

Для начала создаем функцию **read\_maze**, которая на вход получает текстовый файл с лабиринтом, а возвращает его в виде двумерного массива **maze**.

Далее, создаем функцию **get****\_neighbors**, которая на вход получает сам лабиринт **maze** и возвращает список **valid\_neighbors**, состоящий из кортежей, в которых хранятся координаты соседних ячеек, в которые можно перейти.

## a)Поиск в ширину

* Создаем функцию **find\_path**.
* Входным параметром функции является сам лабиринт **maze**.
* В начале задаются точки старта и конца поиска в переменных **start** и **end** соответственно.
* Затем создаётся очередь **queue**, в которую добавляется кортеж **(start, [start])**. В этом кортеже первый элемент — координаты начальной точки, а второй — путь до неё.
* Далее создаётся множество **visited**, в которое будут добавляться посещённые точки.
* Запускается цикл, который продолжается до тех пор, пока очередь **queue** не станет пустой.
* На каждой итерации цикла извлекается из очереди первый элемент **current** и путь **path** до него.
* Если **current** равен точке **end**, значит путь найден и он возвращается.
* Иначе точка **current** добавляется в **visited**, и для каждого соседа **neighbor** точки **current** вызывается функция **get\_neighbors**.
* Для каждого из этих соседей проверяется, что он ещё не посещён. Если это так, то в очередь **queue** добавляется кортеж **(neighbor, path + [neighbor])**, где **neighbor** — это координаты соседней точки, а **path + [neighbor]** — это путь до неё.
* Поиск заканчивается, если весь лабиринт пройден, но если не была найдена точка **end**, в таком случае функция возвращает **None**.

## б)Алгоритм А\*

1.

Создаем функцию **get\_heuristic**, которая вычисляет эвристическое расстояние от текущей ячейки до конечной точки в алгоритме A\*.

Входными параметрами функции являются координаты текущей ячейки **cell** и координаты конечной точки **end**.

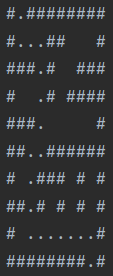
Функция вычисляет расстояние между текущей ячейкой и конечной точкой по формуле Евклида: sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2), где **cell[0]** и **cell[1]** — координаты текущей ячейки, а **end[0]** и **end[1]** — координаты конечной точки. (Эвристическое расстояние используется для оценки стоимости пути от текущей ячейки до конечной точки. Оно добавляется к фактической стоимости пути, чтобы получить общую стоимость пути от начальной точки до конечной. В результате A\* выберет путь с наименьшей общей стоимостью.)

2.

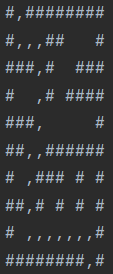
* Создаем функцию **find\_path\_a\_star**.
* Входным параметром функции является сам лабиринт **maze**.
* Сначала создаются начальная и конечная точки **start** и **end**. Затем создается приоритетная очередь **queue**, в которую добавляется кортеж, содержащий стоимость пути, текущую точку и путь к текущей точке.
* Далее создается множество **visited**, в котором будут храниться посещенные точки, чтобы избежать повторных посещений. В цикле **while**, пока очередь не пуста, извлекается кортеж с минимальной стоимостью из приоритетной очереди.
* Затем проверяется, является ли текущая точка конечной точкой. Если да, то функция возвращает стоимость пути и путь к конечной точке.
* Если текущая точка не является конечной точкой, то она добавляется в множество **visited**, чтобы не посещать ее повторно. Далее для каждого соседа текущей точки вызывается функция **get\_neighbors**.
* Если соседняя точка не была посещена ранее, создается новый путь к этой точке, добавляя ее в путь к текущей точке. Затем вычисляется стоимость нового пути, используя формулу **priority = len(new\_path) + get\_heuristic(neighbor, end)**. Здесь **len(new\_path)** - это фактическая стоимость пути от начальной точки до текущей, а **get\_heuristic(neighbor, end)** - это эвристическое расстояние от соседней точки до конечной точки. Общая стоимость нового пути вычисляется как сумма фактической стоимости пути и эвристического расстояния.
* Наконец, создается новый кортеж с общей стоимостью, соседней точкой и новым путем, и добавляется в приоритетную очередь. Это продолжается до тех пор, пока конечная точка не будет достигнута, или пока приоритетная очередь не опустеет.
* Если конечная точка не может быть достигнута из начальной точки, функция возвращает **None**.

# Пример работы

## a)Поиск в ширину



## б)Алгоритм А\*



# Заключение

В ходе проделанной работы были реализованы два алгоритма обхода графа, такие как: Поиск в ширину и А\*.

В результате была разработана программа, которая на выходе создает 2 текстовых(.txt) файла, которые содержат путь от начальной координаты до конечной:

* нарисованный точками “.” – поиск в ширину.
* нарисованный запятыми “,” – А\*.

# Список литературы

1. Дольников, В. Л. Основные алгоритмы на графах : текст лекций / В. Л. Дольников, О. П. Якимова; Яросл. гос. ун-т им. П. Г. Демидова. – Ярославль : ЯрГУ, 2011. – 80 с. (Дата обращения: 25.04.2023).
2. Алексеев В.Е. Таланов В.А. Графы Модели вычислений Структуры данных. / Издательство Нижегородского госуниверситета. – Нижний Новгород: 2004 – 4 c. – Текст: непосредственный. (Дата обращения: 01.05.2023).
3. Рассел, С., Норвиг, П. (2009). Artificial Intelligence: A Modern Approach (3-е изд.). Москва: ДМК Пресс. (Дата обращения: 01.05.2023).

# Приложение 1

## Листинг программы

from queue import PriorityQueue  
from math import sqrt  
  
def read\_maze(filename):  
 *"""  
 Считывает лабиринт из текстового файла и возвращает его в виде двумерного массива  
 """* with open(filename) as f:  
 maze = [[char for char in line.strip()] for line in f]  
 return maze  
  
  
def get\_neighbors(maze, cell: tuple[int, int]):  
 *"""  
 Возвращает список соседних ячеек, в которые можно перейти  
 """* 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  
  
#Поиск в ширину  
def find\_path(maze):  
 *"""  
 Ищет путь от начальной точки до конечной точки в лабиринте  
 """* start = (0, 1)  
 end = (len(maze) - 1, len(maze[0]) - 2)  
  
 queue = [(start, [start])]  
 visited = set()  
 while queue:  
 current, path = queue.pop(0)  
 if current == end:  
 return path  
 visited.add(current)  
 for neighbor in get\_neighbors(maze, current):  
 if neighbor not in visited:  
 queue.append((neighbor, path + [neighbor]))  
 return None  
  
#А\*  
def get\_heuristic(cell, end):  
 *"""  
 Вычисляет эвристическое расстояние от ячейки до конечной точки  
 """* return sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2)  
  
def find\_path\_a\_star(maze):  
 *"""  
 Ищет путь от начальной точки до конечной точки в лабиринте, используя алгоритм A\*  
 """* start = (0, 1)  
 end = (len(maze) - 1, len(maze[0]) - 2)  
 queue = PriorityQueue()  
 queue.put((0, start, [start]))  
 visited = set()  
 while not queue.empty():  
 p, current, path = queue.get()  
 if current == end:  
 return p, path  
 visited.add(current)  
 for neighbor in get\_neighbors(maze, current):  
 if neighbor not in visited:  
 new\_path = path + [neighbor]  
 priority = len(new\_path) + get\_heuristic(neighbor, end)  
 queue.put((priority, neighbor, new\_path))  
 return None  
  
  
def main():  
 filename = "maze-for-u.txt"  
 maze = read\_maze(filename)  
 """  
 Реализация создания выходного файла для поиска в ширину  
 """  
 path1 = find\_path(maze)  
   
 for place in path1:  
 maze[place[0]][place[1]] = "."  
  
 result1 = ""  
 for line in maze:  
 result1 += "".join(line) + "\n"  
  
 with open("res1.txt", "w") as f:  
 f.write(result1)  
  
 """  
 Реализация создания выходного файла для А\*  
 """  
 path2 = find\_path\_a\_star(maze)  
 path22 = path2[1]  
  
 for place in path22:  
 maze[place[0]][place[1]] = ","  
  
 result2 = ""  
 for line in maze:  
 result2 += "".join(line) + "\n"  
  
 with open("res2.txt", "w") as f:  
 f.write(result2)  
main()