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 Скворцов Д.В.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc1)

[Задачи 3](#_Toc2)

[1.Теоретическая часть 4](#_Toc3)

[2. Реализация алгоритма 6](#_Toc4)

[Пример работы 7](#_Toc5)

[Заключение 8](#_Toc6)

[Список литературы 9](#_Toc7)

[Приложение 1 10](#_Toc8)

[Листинг программы 10](#_Toc9)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании [1]. Графы являются существенным элементом математических моделей в самых разнообразных областях науки и практики. Они помогают наглядно представить взаимоотношения между объектами или событиями в сложных системах [2].

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

# Задачи

* Изучить алгоритмы построения маршрута в графе;
* Выделить особенности, необходимые в конкретной задаче поиска маршрута;
* Подготовить исходные данные: лабиринт и исходные координаты точек;
* Написать алгоритмы с заданными параметрами;
* Сохранить результаты обходов лабиринта и маршруты в файл.

# 1.Теоретическая часть

Обход в ширину является основой для многих важных алгоритмов для работы с графами. Далее приводится базовый алгоритм обхода графа в ширину (рис. 7.8). На определенном этапе каждая вершина графа переходит из состояния неоткрытая в состояние открытая. При обходе в ширину неориентированного графа каждому ребру присваивается направление: от открывающей вершины и к открываемой вершине у. В этом контексте вершина и называется родителем, или предшественником, вершины у, а вершина у — потомком вершины и. Поскольку каждый узел, за исключением корня, имеет только одного родителя, получится дерево вершин графа — оно и определяет кратчайший путь от корня ко всем другим узлам графа. Это свойство делает обход в ширину очень полезным в решении задач поиска кратчайшего пути. Ребра графа, которые не включены в дерево обхода в ширину, также имеют особые свойства. Для неориентированных графов не попавшие в дерево ребра могут указывать только на вершины на том же уровне, что и родительская вершина, или на вершины, расположенные на уровень ниже. Эти свойства естественно следуют из того факта, что каждое ребро в дереве должно быть кратчайшим путем в графе. А для ориентированных графов ребро (м, у), указывающее в обратном направлении, может существовать в любом случае, когда вершина у расположена ближе к корню, чем вершина

Реализация:

Процедура обхода в ширину Е использует два массива булевых значений для хранения информации о каждой вершине графа. Вершина открывается при первом ее посещении. Когда все исходящие из вершины ребра были исследованы, то вершина считается обработанной. Таким образом, как уже отмечалось ранее, в процессе обхода состояние каждой вершины начинается с неоткрытого, переходит в открытое и заканчивается обработанным. Эту информацию можно было бы хранить с помощью одной переменной перечислимого типа, но мы используем две булевы переменные:



Многие элементарные алгоритмы для работы с графами выполняют один или два обхода графа, в процессе которых они выполняют какие-либо действия. Любой из таких алгоритмов, если он корректно реализован с использованием списков смежности, обязательно имеет линейное время исполнения, поскольку обход в ширину исполняется за

время О(n+ m) как на ориентированных, так и на неориентированных графах. Это, оптимальное время, потому что именно за такое время можно лишь прочитать граф из n вершин и m ребер.

Секрет мастерства заключается в умении видеть ситуации, в которых применение таких обходов гарантированно даст положительные результаты.

Граф называется связным (connected), если имеется путь между любыми двумя его вершинами, Связность графа дружеских отношений означает, что любые два человека в нем связаны цепочкой из людей, попарно знакомых друг с другом.

Компонентом связности неориентированного графа называется максимальный набор его вершин, для которого существует путь между каждой парой вершин. Эти компоненты являются отдельными «кусками» графа, которые не соединены между собой. В качестве примера отдельных компонентов связности в графе дружеских отношений можно привести обитающие где-то в джунглях первобытные племена, которые еще не были открыты для остального мира. А отшельник в пустыне или крайне неприятный человек будет примером компонента связности, состоящего из одной вершины.

Удивительно, какое большое количество кажущихся сложными проблем сводится к поиску или подсчету компонентов связности. Например, вопрос, можно ли решить какую-нибудь головоломку (скажем, кубик Рубика), начав с определенной позиции, по сути, представляет собой вопрос, является ли связным граф возможных конфигураций

Компоненты связности можно найти с помощью обхода в ширину, т. к. порядок перечисления вершин не имеет значения. Начнем с выполнения поиска, производя его от произвольной вершины. Все элементы, обнаруженные в процессе этого обхода, должны быть членами одного и того же компонента связности. Потом повторим обход, начиная с любой неоткрытой вершины (если таковая имеется), чтобы определить второй компонент связности, — и т.д. до тех пор, пока не будут обнаружены все вершины[2].

# 2. Реализация алгоритма

Стандартная реализация ВFS помещает каждую вершину графа в одну из двух категорий:

1. Посещенные.
2. Не посещенные.

*Цель алгоритма*- пометить каждую вершину, как посещенную, избегая циклов.

Алгоритм реализован следующим образом:

1. Определение начальной и конечной точек.
2. Создание очереди для хранения узлов, которые нужно обработать.
3. Создание словаря для хранения информации о каждом узле: его родительском узле и расстоянии от начальной точки.
4. Добавление начальной точки в очередь и установка ее расстояния от самой себя равным 0.
5. Пока очередь не пуста, извлечение первого узла из очереди и проверка, является ли он конечной точкой.
6. Если узел является конечной точкой, восстановление пути от начальной точки до конечной точки, используя информацию из словаря.
7. Если узел не является конечной точкой, добавление всех его соседей в очередь, если они еще не были посещены, и обновление информации о них в словаре.
8. Повторение шагов 5-7 до тех пор, пока не будет найден путь от начальной точки до конечной точки или пока очередь не опустеет.

Граф может иметь две разные несвязанные части, поэтому, чтобы убедиться, что мы покрываем каждую вершину, мы также можем запустить алгоритм BFS на каждом узле.

# Пример работы



Точками обозначается путь по лабиринту до A\*



Запятыми обозначается путь по лабиринту от A\* до выхода

# Заключение

В ходе проделанной работы были реализован алгоритм обхода графа: поиск в ширину и A\* для задачи поиска маршрута в лабиринте.

В результате было разработано консольное приложение для поиска выхода из лабиринта через ключ.

# Список литературы

1. Скиена С. Стивен Алгоритмы. Руководство по разработке. 3-е изд / Скиена С. Стивен. – СПб.: Springer, 2022. – 250 - 257, 848 с. – Текст:  
   непосредственный.
2. Алексеев В.Е. Таланов В.А. Графы Модели вычислений Структуры данных. / Издательство Нижегородского госуниверситета. – Нижний Новгород: 2004 – 4 c. – Текст: непосредственный.
3. OpenAI ChatGPT [электронный ресурс] - <https://openai.com/blog/chatgpt> (Дата обращения: 22.04.2023)

# Приложение 1

## Листинг программы

from queue import PriorityQueue

from math import sqrt

import random

def read\_maze(filename):

with open(filename) as f:

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

return maze

#создает ключ в рандомном месте

height = len(read\_maze("maze-for-u.txt"))

width = len(read\_maze("maze-for-u.txt")[0])

passages = []

for i in range(height):

for j in range(width):

if read\_maze("maze-for-u.txt")[i][j] == " ":

passages.append((i, j))

random\_key = random.choice(passages)

def get\_neighbors(maze, cell: tuple[int, int]):

#соседи

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):

start = (0, 1)

key = random\_key

queue = [(start, [start])]

visited = set()

while queue:

current, path = queue.pop(0)

if current == key:

return path

visited.add(current)

for neighbor in reversed(get\_neighbors(maze, current)):

if neighbor not in visited:

queue.append((neighbor, path + [neighbor]))

return None

# А\*

def get\_heuristic(cell, end):

#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):

key = random\_key

end = (len(maze) - 1, len(maze[0]) - 2)

queue = PriorityQueue()

queue.put((0, key, [key]))

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)

#Создание текстового документа, который рисует путь точками от входа до ключа и запятыми от ключа до выхода

path1 = find\_path(maze)

path2 = find\_path\_a\_star(maze)

path22 = path2[1]

for place in path1:

maze[place[0]][place[1]] = "."

result1 = ""

for line in maze:

result1 += "".join(line) + "\n"

for place in path22:

maze[place[0]][place[1]] = ","

result2 = ""

for line in maze:

result2 += "".join(line) + "\n"

with open("result.txt", "w") as f:

f.write(result2)

main()