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: «Разработка программы для поиска пути в Лабиринте»

Выполнил студент 1 курса гр. ИС-34 Королев Н.Д.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Оглавление

[Введение 3](#_Toc133530358)

[Задачи 3](#_Toc133530359)

[1.Теоретическая часть 4](#_Toc133530360)

[1.1 Графы 4](#_Toc133530361)

[1.2 Алгоритмы построения маршрута в графе, используемые в разработке 4](#_Toc133530362)

[**1.3 Особенности реализации** 6](#_Toc133530363)

[2. Реализация алгоритма 7](#_Toc133530364)

[2.1 Подготовительный этап 7](#_Toc133530365)

[2.2 Функции для нахождения пути 7](#_Toc133530366)

[2.3. Построение маршрута 8](#_Toc133530367)

[Пример работы 9](#_Toc133530368)

[Заключение 10](#_Toc133530369)

[Список литературы 11](#_Toc133530370)

[Приложение 1 12](#_Toc133530371)

[Листинг программы 12](#_Toc133530372)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании [1]. Алгоритмы обхода графа появились позже самой теории графов, самими же графами интересовались довольно таки давно. Швейцарский, прусский и российский математик Леонард Эйлер в статье (на латинском языке, изданной Петербургской академией наук) о решении знаменитой задачи о кёнигсбергских мостах, датированной 1736 годом, первым применил идеи теории графов при доказательстве некоторых утверждений. При этом Эйлер не использовал ни сам термин «граф», ни какие-либо термины теории графов, ни изображения графов. Леонард Эйлер считается отцом теории графов (как и топологии), открывшим понятие графа, а 1736 год назначен годом рождения теории графов [4]. Одним из таких является поиск в ширину [2].

**Цель работы**: реализовать алгоритмы обхода графа в ширину 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.
* Предоставить листинг программы, а также результат работы на тестовом графе

# 1.Теоретическая часть

## 1.1 Графы

Стоит начать с того, Что же вообще такое граф?

**Граф** — математическая абстракция реальной системы любой природы, объекты которой обладают парными связями. Граф как математический объект есть совокупность двух множеств — множества самих объектов, называемого множеством вершин, и множества их парных связей, называемого множеством рёбер. Элемент множества рёбер есть пара элементов множества вершин. [5]

Стоит отметить, что в данной работе автор обращается к графу, имеющему двумерный вид, то бишь граф по 2 координатам, данный граф можно встретить во многих учебниках, в программе Excel и во многих других местах. Граф же для работы программы был преобразован из текстового файла содержащего 3 условных символа: « » - свободное пространство, по которому можно перемещаться в пределах самого лабиринта, «#» - стена, через которую нельзя проходить и «\*» - ключ, условное место, находящееся в свободном пространстве, через которое обязательно надо пройти.

Непосредственно код принимает данный граф как двумерный массив, заполненный символами « », «#» и «\*».

## 1.2 Алгоритмы построения маршрута в графе, используемые в разработке

Во Время работы на д кодом автор использовал 2 алгоритма построения маршрута в графе: Поиск в ширину и А\*, поиск в ширину искал путь до ключа, особой точки, которую нужно пройти перед тем, как попасть на выход, поиском же до выхода занимался алгоритм А\*.

Обход или поиск — это одна из фундаментальных операций, выполняемых на графах. **Поиск в ширину**начинается с определённой вершины, затем исследуются все её соседи на данной глубине и происходит переход к вершинам следующего уровня. В графах, в отличие от деревьев, могут быть циклы — пути, в которых первая и последняя вершины совпадают. Поэтому необходимо отслеживать посещённые алгоритмом вершины. При реализации алгоритма поиска в ширину используется структура данных «очередь». [6]

Плюсы и минусы данного алгоритма:

Пространственная сложность

Так как в памяти хранятся все развёрнутые узлы, пространственная сложность алгоритма составляет �(|�|+|�|).

Алгоритм поиска с итеративным углублением похож на поиск в ширину тем, что при каждой итерации перед переходом на следующий уровень исследуется полный уровень новых узлов, но требует значительно меньше памяти.

Временная сложность

Так как в худшем случае алгоритм посещает все узлы графа, при хранении графа в виде списков смежности, временная сложность алгоритма составляет �(|�|+|�|).

Полнота

Если у каждого узла имеется конечное число преемников, алгоритм является полным: если решение существует, алгоритм поиска в ширину его находит, независимо от того, является ли граф конечным. Однако если решения не существует, на бесконечном графе поиск не завершается.

Оптимальность

Если длины рёбер графа равны между собой, поиск в ширину является оптимальным, то есть всегда находит кратчайший путь. В случае взвешенного графа поиск в ширину находит путь, содержащий минимальное количество рёбер, но не обязательно кратчайший.

Поиск по критерию стоимости является обобщением поиска в ширину и оптимален на взвешенном графе с неотрицательными весами рёбер. Алгоритм посещает узлы графа в порядке возрастания стоимости пути из начального узла и обычно использует очередь с приоритетами. [7]

Именно из-за оптимальности был выбран этот способ, ибо ребра между элементами графа равны между собой, а сам код написан так, чтобы не хранить в памяти «тупиковые ветви», это потом понадобиться в построении пути.

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, *весь* пройденный до неё путь. Составляющая *g(x)* — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение *f(x)*, после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в очереди с приоритетом. Приоритет пути определяется по значению *f(x) = g(x) + h(x)*. Алгоритм продолжает свою работу до тех пор, пока значение *f(x)* целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью. [8]

Данный алгоритм был выбран для того, чтобы посмотреть его работу в условиях двумерного массива, где он и нашёл оптимальный путь. С помощью него был построен путь от точки ключа до выхода, так же его можно было бы применить для поиска пути от любой до любой точки в графе.

**1.3 Особенности реализации**

После того, как граф был преобразован из текстового файла в массив, автор решил использовать аватара. У аватара есть точка нахождения в данный момент, соседи, а также пройденный им путь. Для алгоритма поиска в ширину нужны были лишь координаты аватара, ключа и соседних точек. Для алгоритма А\* нужны были координаты ключа, выхода а также прямой путь от ключа до выхода, придерживаясь которого и будет построен путь до выхода. Т.к. ребра графа имеют единичную длину, а путь всего один, то разницы между ними будет не много, в ином же варианте, если бы ребра были разных длин, то алгоритм А\* показал бы себя намного лучше.

Также предстоит создать файл, в котором будет построен путь в виде точек до ключа и запятыми от ключа до выхода, в некоторых местах путь будет повторяться, в таком случае он будет обозначен запятыми, ибо путь от ключа до выхода был построен позже, чем от входа до ключа. Сам конечный файл это просто измененная копия изначального файла, но можно было бы и создать его из программы соответственно, что автоматизировало бы процесс.

Средой разработки была выбрана программа PyCharm, ибо для автора она была интуитивно понятней, чем любая другая. Язык программирования – Python 3.11. Удобен в освоении, соответствует дисциплине и имеет набор необходимых функций.

# 2. Реализация алгоритма

## 2.1 Подготовительный этап

Для начала работы нужно подготовить сам файл, текстовый файл. Сгенерировать его можно с помощью других программ, такой файл и был взят за основу. Более ничего из сторонних файлов нам не нужно.

В самой программе была написана функция ReadMaze которая вернула нам готовый к обработке двумерный массив. Для корректного поиска пути нам нужно определить где вход, а где выход. Исходя из того, что границы самого лабиринта содержат вход и выход, там и нужно их искать, эти координаты находит функция StartEnd. После нахождения входа и выхода надо найти точку ключа, это будет легко, ибо ключ всего лишь один на весь массив, и имеет уникальное обозначение: «\*», эти занимается функция Key.

## 2.2 Функции для нахождения пути

Для того, чтобы начать поиск пути – нужно написать функцию для аватара, которая будет сообщать ему, в какие точки можно будет идти в данный момент, для этого была создана shag, которая возвращает координаты доступных для посещения ячеек в лабиринте. Для нахождения того самого единственного пути в лабиринте была сделана функция pathtoend которая возвращает нам координаты пути от точки входа до точки выхода, нужно это будет в будущем для построения маршрута вход->ключ. На этом этапе на основе pathtoend можно так же сделать функцию pathstartkey, она нам понадобиться в конце, когда мы будем строить точками путь до ключа от входа.

Теперь можно приступить к реализации поиска маршрута от входа до точки ключа с помощью алгоритма обхода в ширину. Ранее был найден путь от входа до выхода, который понадобиться нам на этом этапе. Если учитывать, что путь всего один, то самый оптимальный путь можно найти, если идти не от какой то точки на маршруте до ключа, а от ключа до самой ближней точки маршрута. На основе предыдущих функций была создана pathfromkey, ее единственное отличие в том, что начальная точка принимается за точку ключа, а конечная – первая точка из маршрута вход -> выход.

Здесь описываются функции, который лишь предлагают альтернативный путь к нахождению пути до ключа и работает конкретно на поиск по ширине до ключа. MakeFPAth прописывает координаты пути от входа до точки , откуда идет отклонение от маршрута вход->выход. Функция MakeSPath же достраивает путь, который был начат в MakeFPath, далее это функция не несет в себе смысла, просто представлена в качестве альтернативы поиска пути найденному в pathstartkey.

Далее будет найден путь от ключа до выхода с помощью алгоритма A\*. Для начала автор сделал функцию heur, которая находит эвристическое расстояние от одной точки до другой, в данном примере нам понадобится точка ключа как начало и точка выхода как конец. Функция star находит оптимальный маршрут с помощью heur, как бы опираясь на тот путь, что она построила, далее функция star будет использована для построения маршрута ключ->выход в конечном файле.

## 2.3. Построение маршрута

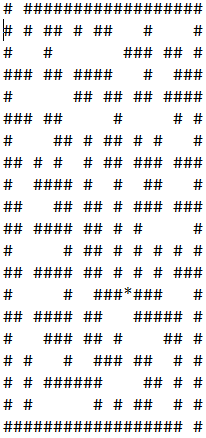
Конечным продуктом программы должен стать файл с построенным маршрутом вход->ключ в виде точек а также ключ-> выход в виде запятых, именно для такой цели служит функция main, как бы собирая воедино все ранее написанные функции, кроме MakeSPath.

Сама main может служить только для этой программы, для использования в других проектах – нужно будет переписывать несколько параметров ней, такие как: исходный текстовый файл и итоговый файл с результатом.

Построение пути состоит из 3 этапов: построить путь вход->ключ, построить путь ключ->выход а также вновь отметить точку ключа с помощью «\*». Точка отмечается, ибо при построении пути она была перекрыта маршрутом. Для построения нужно сначала создать и открыть конечный файл, в который и будут заносится изменения. Путь вход->ключ будет строиться с помощью функции pathstartkey, путь ключ->выход будет строиться при помощь функции star а точка ключа вновь проставляться при помощи функции Key. В конце остается лишь занести изменения в конечный файл и все, готово.

# Пример работы

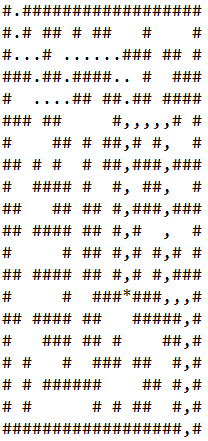
Изначальный файл: “xtt.txt” в версии 3 лабораторной: “maze-for-u.txt”



Ключ, до которого нужно добраться аватару

Конечный файл: “txxt.txt” в версии 3 лабораторной: “maze-for-me-done.txt”

Точками отмечен путь до ключа, маршрут начало-ключ



Запятыми отмечен путь от ключа до выхода, так же перекрывает «путь из точек», т.к. этот маршрут был проложен позже маршрута начало-ключ

Ключ

# Заключение

В ходе проделанной работы была создана программа для поиска пути в лабиринте, поиск пути в котором был осуществлен с помощью алгоритмов обхода графа в ширину и А\*.Разработка программы потребовала изучение таких алгоритмов, как Обход графа в ширину и А\*.

В результате был разработан код на языке программирования Python 3.11, который работает с текстовыми файлами, а также с из копиями, для постройки пути в лабиринте.

# Список литературы

1. Скиена С. Стивен Алгоритмы. Руководство по разработке. 3-е изд / Скиена С. Стивен. – СПб.: Springer, 2022. – 848 с. – Текст:  
   непосредственный.
2. Агапов И., «Обход графа: поиск в глубину и поиск в ширину простыми словами на примере JavaScript» [Электронный ресурс]. URL: <https://habr.com/ru/articles/504374/> (Дата обращения: 11.04.2023).
3. Станковец А.В. АЛГОРИТМЫ НА ГРАФАХ / Станковец А.В. – Текст: электронный // M[ODERN SCIENCE](https://www.elibrary.ru/contents.asp?id=44150018). — 2020. — №10-2. — С.532-536. — EDN: [pdmxpx](https://www.elibrary.ru/pdmxpx)
4. «Теория графов» / [Электронный ресурс] // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/Теория\_графов (дата обращения: 27.04.2023).
5. «Граф (математика)» / [Электронный ресурс] // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/Граф\_(математика) (дата обращения: 27.04.2023).
6. пользователь Андрей Шагин "10 графовых алгоритмов" / пользователь Андрей Шагин [Электронный ресурс] // Medium : [сайт]. — URL: https://medium.com/nuances-of-programming/10-графовых-алгоритмов-наглядное-объяснение-53226d8e6ba0 (дата обращения: 27.04.2023).
7. "Поиск в ширину" / [Электронный ресурс] // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/Поиск\_в\_ширину (дата обращения: 27.04.2023).
8. "A\*" / [Электронный ресурс] // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/A\* (дата обращения: 27.04.2023).

# Приложение 1

## Листинг программы

from math import sqrt  
from queue import PriorityQueue  
  
def ReadMaze(file):  
 with open(file, "r") as f:  
 lns = f.read().splitlines()  
 maze = [list(line) for line in lns]  
 return maze  
  
def StartEnd(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 = (int(len(maze)-1),int(i))  
 return start, end  
  
def 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 shag(maze, Cord):  
 glb, dln = Cord  
 neig = [(glb-1,dln),(glb,dln+1),(glb+1,dln),(glb,dln-1)]  
 shagi = []  
 for nei in neig:  
 glb,dln = nei  
 if 0<=glb<len(maze) and 0<=dln<len(maze[0]) and maze[glb][dln]!="#":  
 shagi.append(nei)  
 return shagi  
  
def pathtoend(RM):  
 path=[]  
 start,end = StartEnd(RM)  
 queue = [(start, [start])]  
 visited=[]  
 av = start  
 while av != end:  
 av, path = queue.pop(0)  
 visited.append(av)  
 for cells in shag(RM,av):  
 if cells not in visited:  
 queue.append((cells, path + [cells]))  
 return path  
  
def pathstartkey(RM):  
 path=[]  
 start,end = StartEnd(RM)  
 end = Key(RM)  
 queue = [(start, [start])]  
 visited=[]  
 av = start  
 while av != end:  
 av, path = queue.pop(0)  
 visited.append(av)  
 for cells in shag(RM,av):  
 if cells not in visited:  
 queue.append((cells, path + [cells]))  
 return path  
  
def pathfromkey(RM):  
 pathkey=[]  
 start = Key(RM)  
 queue = [(start,[start])]  
 visited=[]  
 av = start  
 while av not in pathtoend(RM):  
 av, path = queue.pop(0)  
 visited.append(av)  
 for cells in shag(RM,av):  
 if cells not in visited:  
 queue.append((cells, path + [cells]))  
 return path  
  
def MakeFPath(pathtoend,pathfromkey):  
 f = []  
 s = pathfromkey[len(pathfromkey)-1]  
 for i in pathtoend:  
 if i == s:  
 return f  
 else:  
 f.append(i)  
  
def MakeSPath(MakeFPath, pathfromkey):  
 s = pathfromkey[len(pathfromkey)-1]  
 l = []  
 path=[]  
 for k in range(len(pathfromkey)-1,0,-1):  
 l.append(pathfromkey[k])  
 for i in MakeFPath:  
 path.append(i)  
 for i in l:  
 path.append(i)  
 print(path)  
  
def heur(cell,end):  
 return sqrt((cell[0]-end[0]) \*\* 2 + (cell[1]-end[1]) \*\* 2)  
  
def star(RM, st, en):  
 start = st  
 end = en  
 queue = PriorityQueue()  
 queue.put((0, start, [start]))  
 visited = set()  
 while not queue.empty():  
 p, av, path = queue.get()  
 if av == end:  
 return p, path  
 visited.add(av)  
 for cells in shag(RM, av):  
 if cells not in visited:  
 path1 = path + [cells]  
 pr = len(path1) + heur(cells, end)  
 queue.put((pr, cells, path1))  
 return None  
  
def main():  
 RM = ReadMaze("maze-for-u.txt")  
 k = Key(RM)  
 pathtokey = pathstartkey(RM)  
 start, end = StartEnd(RM)  
 PathToEnd = star(RM, Key(RM), end)  
  
 for i in pathtokey:  
 RM[i[0]][i[1]] = "."  
  
 for i in star(RM, k, end)[1]:  
 RM[i[0]][i[1]] = ","  
  
 RM[k[0]][k[1]] = "\*"  
  
 result = ""  
 for line in RM:  
 result += "".join(line) + "\n"  
  
 with open("maze-for-me-done.txt", "w") as f:  
 f.write(result)  
 print("смотрите файл")  
main()