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 Емельянов Д.В.

Проверил Морозов Н.С.

Нижний Новгород 2023 г.

Содержание

[Введение 3](#_Toc136203799)

[1.Теоретическая часть 4](#_Toc136203800)

[Поиск в ширину (breadth-first search, BFS) 4](#_Toc136203801)

[Описание работы алгоритма 4](#_Toc136203802)

[Для чего нужен BFS 5](#_Toc136203803)

[Алгоритм А\* (А Star или А «со звездочкой») 6](#_Toc136203804)

[Как работает алгоритм «A Star» 6](#_Toc136203805)

[2. Реализация алгоритма 9](#_Toc136203806)

[Пример работы 11](#_Toc136203807)

[Заключение 12](#_Toc136203808)

[Список литературы 13](#_Toc136203809)

[Приложение 1 14](#_Toc136203810)

[Листинг программы 14](#_Toc136203811)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании.

Поиск пути - определение компьютером самого оптимального маршрута между двумя точками. Чаще всего можно столкнуться при разработке игр, разработке роботов-манипуляторов, например, если в них есть свободное пространство. Компьютеру нельзя просто сказать куда нужно идти, что нужно сделать. Разработчику необходимо указать чёткий набор команд (найти синус угла между соседними клетками, повернуть персонажа(манипулятор) на угол, переместиться на 3 метра) или указать полный путь по точкам от A до Z. Но ведь разработчик не может написать один паттерн поведения для всех случаев, нам необходимо чтобы программа сама находила промежуточные точки и шла по ним. Иногда задача должна быть решена идеально, т.е. программа обязана найти лучший путь.

В данной работе я рассматривал множество способов, однако применил лишь два: Алгоритм поиска в ширину (Breadth-first search) и алгоритм A\*.

**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.

# 1.Теоретическая часть

## Поиск в ширину (breadth-first search, BFS)

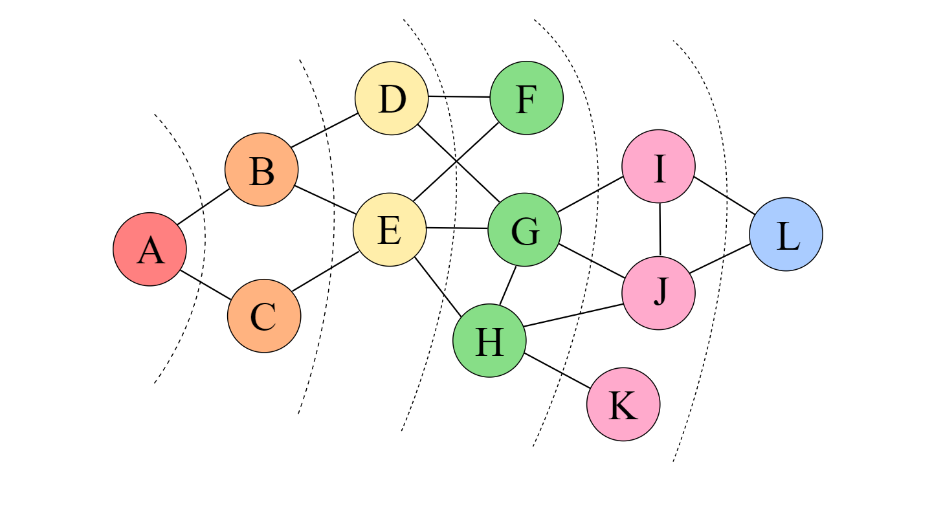
Алгоритм был разработан независимо Муром и Ли для разных приложений (поиск пути в лабиринте и разводка проводников соответственно) в 1959 и 1961 годах. Этот алгоритм можно сравнить с поджиганием соседних вершин графа: сначала мы зажигаем одну вершину (ту, из которой начинаем путь), а затем огонь за один элементарный промежуток времени перекидывается на все соседние с ней не горящие вершины. В последствие то же происходит со всеми подожженными вершинами. Таким образом, огонь распространяется «в ширину». В результате его работы будет найден кратчайший путь до нужной клетки [2].

## Описание работы алгоритма

На вход алгоритма подаётся невзвешенный граф и номер стартовой вершины s. Граф может быть как ориентированным, так и неориентированным — для алгоритма это не важно.

Основную идею алгоритма можно понимать как процесс «поджигания» графа: на нулевом шаге мы поджигаем вершину s, а на каждом следующем шаге огонь с каждой уже горящей вершины перекидывается на всех её соседей, в конечном счете поджигая весь граф.

Если моделировать этот процесс, то за каждую итерацию алгоритма будет происходить расширение «кольца огня» в ширину на единицу. Номер шага, на котором вершина v начинает гореть, в точности равен длине её минимального пути из вершины s.



Моделировать это можно следующим образом. Создадим очередь, в которую будут помещаться горящие вершины, а также заведём булевый массив, в котором для каждой вершины будем отмечать, горит она или нет — или иными словами, была ли она уже посещена. Изначально в очередь помещается только вершина s, которая сразу помечается горящей. Затем алгоритм представляет собой такой цикл: пока очередь не пуста, достать из её головы одну вершину v, просмотреть все рёбра, исходящие из этой вершины, и если какие-то из смежных вершин u ещё не горят, поджечь их и поместить в конец очереди. В итоге, когда очередь опустеет, мы по одному разу обойдём все достижимые из s вершины, причём до каждой дойдём кратчайшим путём. Длины кратчайших путей можно посчитать, если завести для них отдельный массив d и при добавлении в очередь пересчитывать по правилу *du*​=*dv*​+1. Также можно компактно сохранить дополнительную информацию для восстановления самих путей, заведя массив «предков», в котором для каждой вершины хранится номер вершины из которой мы в неё попали [7].

## Для чего нужен BFS

1. Для решения задач поиска оптимального пути. Классической задачей считается автоматизированный поиск выхода из лабиринта.
2. Для решения задач, связанных непосредственно с теорией графов, например для поиска компонент связности. Эти задачи в свою очередь решаются в Data Science, теории сетей и электронике.
3. Для задач искусственного интеллекта, связанных с поиском решения с минимальным количеством ходов. В таком случае состояния «умной машины» представляются как вершины, а переходы между ними — как ребра.
4. Для оптимизации памяти при обходе графа в некоторых ситуациях, например для некоторых специфических структур.
5. Для работы с информацией в определенных структурах данных, таких как деревья. Их тоже можно обходить с помощью алгоритма BFS, потому что они — подвид графов [4].

## Алгоритм А\* (А Star или А «со звездочкой»)

В 1964 году Нильс Нильсон изобрел эвристический подход к увеличению скорости алгоритма Дейкстры. Этот алгоритм был назван А1. В 1967 году Бертрам Рафаэль сделал значительные улучшения по этому алгоритму, но ему не удалось достичь оптимальности. Он назвал этот алгоритм A2. Тогда в 1968 году Петр Э. Харт представил аргументы, которые доказывали, что A2 был оптимальным при использовании последовательной эвристики лишь с незначительными изменениями. В его доказательство алгоритма также включен раздел, который показывал, что новый алгоритм A2 был, возможно, лучшим алгоритмом, учитывая условия.

Алгоритм A\* на данный момент является оптимальным способом поиска пути между двумя точками в тех случаях, когда существует сравнительно простой эвристический метод оценки расстояния между элементами области поиска. Если такого метода не существует, A\* идентичен либо алгоритму Дейкстры в вариации для двух точек, либо волновому алгоритму в зависимости от вида области поиска.

Также алгоритм A\* не оптимален, если область поиска статична и поиск пути на ней осуществляется множество раз, поскольку в таком случае все пути можно заранее рассчитать при помощи алгоритма Дейкстры для всех точек [2].

## Как работает алгоритм «A Star»

Посетив одну конкретную вершину, алгоритм «A Star» перед переходом к следующей исследует все соседние вершины. Все вершины алгоритм разделяет на 3 категории:

1. 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.
2. 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.
3. Исследованные вершины. В эту категорию попадают те вершины, которые уже были посещены алгоритмом «A Star». 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 Star» 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 Star» находит кратчайший путь между вершинами, основываясь на стоимости и «весе» ребер. Поэтому путь, который находит «A Star», можно по праву назвать «самым быстрым» или «самым простым». По этой причине алгоритм «A Star» очень часто применяется как раз для планирования кратчайших путей, его также часто применяют в играх.

Однако его применение ограничивается его главным недостатком — потребностью в большом количестве памяти, потому что алгоритм «A Star» хранит всю информацию об известных и исследованных вершинах [7].

# 2. Реализация алгоритма

Функция **read\_maze** читает файл filename и возвращает двумерный список (матрицу), представляющую лабиринт. Функция использует конструкцию генератора списка для создания списка символов в каждой строке файла.

Функция **bfs** использует алгоритм поиска в ширину (BFS) для поиска кратчайшего пути в лабиринте, заданном в виде двумерного списка. Функция начинает поиск с начальной точки и заканчивает поиск на клетке, которая является точкой выхода из лабиринта. Проверяется является ли эта точка конечной. Если да, то показывается путь до этой точки, если нет, то данная точка добавляется в список посещенных вершин и аватар идет дальше. Если пусть до конечной не найден, то функция возвращает None.

**get\_heuristic** - это функция, которая принимает координаты двух клеток и возвращает евклидово расстояние между ними. Она используется в алгоритме A\* для оценки расстояния от текущей клетки до конечной клетки.

Функция **get\_neighbors** принимает матрицу лабиринта и координаты клетки, затем возвращает список соседних клеток, которые можно достичь из данной клетки. Функция проверяет, что соседние клетки находятся внутри границ лабиринта и не являются стенами.

Функция **solve\_maze** использует алгоритм поиска в ширину (BFS) для поиска кратчайшего пути в лабиринте, в заданном в файле "maze-for-u.txt". Функция сначала считывает лабиринт из файла, затем вызывает функцию bfs. После этого функция изменяет лабиринт, заменяя все клетки на пути к выходу на символ «.». Затем результат записывается в файл «maze-for-me-done1.txt». Потом используется алгоритм A\* для поиска кратчайшего пути в лабиринте, заданном в файле «maze-for-u.txt». Функция сначала считывает лабиринт из файла, затем вызывает функцию a\_star для поиска кратчайшего пути. После этого функция изменяет лабиринт, заменяя все клетки на пути к выходу на символ «,». Затем результат записывается в файл «maze-for-me-done2.txt». В итоге функция solve\_maze() решает задачу поиска выхода из лабиринта и записывает результат в файлы.

На вход подаётся файл «maze-for-u.txt», сгенерированный программой «gen\_lab\_origin.py». «maze-for-u.txt» - исходный лабиринт, где пробелы являются свободными клетками, по которым может перемещаться аватар, а «#» - являются стенами, сквозь которые проходить нельзя.

В результате работы моего кода (приложение 1), будет создано 2 файла: «maze-for-me-done1.txt» и «maze-for-me-done2.txt» на основе файла «maze-for-u.txt». В файле «maze-for-me-done1.txt» строится путь от входа в лабиринт до его выхода, при помощи «.», используя алгоритм поиска пути в ширину. В файле «maze-for-me-done2.txt» строится путь от входа в лабиринт до его выхода при помощи запятых «,», используя алгоритм А\*.

# Пример работы

«maze-for-me-done1.txt»



«maze-for-me-done2.txt»



# Заключение

В ходе проделанной работы была создана программа для поиска пути в лабиринте. Поиск пути был осуществлен с помощью алгоритмов обхода графа в ширину и А Star. Разработка программы потребовала изучение алгоритмов поиска пути и были реализованы 2 из них: Поиск в ширину, алгоритм A\* (A Star).

# Список литературы

1. Алгоритм поиска A\* // medium.com URL: https://medium.com/nuances-of-programming/алгоритм-поиска-a-3bb59be05a79 (дата обращения: 26.04.2023).
2. Алгоритм A\* // studfile.net // URL: https://studfile.net/preview/3239584/page:4/ (дата обращения: 26.04.2023).
3. Алгоритмы поиска пути в графе, алгоритм BFS // tproger.ru URL: https://tproger.ru/articles/pathfindings/  (дата обращения: 26.04.2023).
4. Алгоритмы поиска в ширину в графе и его приложения // e-maxx.ru URL: https://e-maxx.ru/algo/bfs (дата обращения: 26.04.2023).
5. Алгоритмы. Руководство по разработке. 3-е изд. // anyflip.com URL: https://coderlessons.com/tutorials/kompiuternoeprogrammirovanie/osnovy-algoritmov/1-zhadnyi-algoritm(дата обращения: 26.04.2023).
6. Поисковый алгоритм A Star: что это и как эффективно его использовать? // codernet.ru URL: https://codernet.ru/articles/web/poiskovyij\_algoritm\_a\_star\_chto\_eto\_i\_kak\_effektivno\_ego\_ispolzovat/ (дата обращения: 26.04.2023).
7. Поиск в ширину – алгоритмика // ru.algorithmica.org // URL:

<https://ru.algorithmica.org/cs/shortest-paths/bfs/> (дата обращения: 26.04.2023).

1. BFS // blog.skillfactory.ru URL: https://blog.skillfactory.ru/glossary/bfs/ (дата обращения: 26.04.2023).

# Приложение 1

## Листинг программы

from math import sqrt

from queue import PriorityQueue

def read\_maze(filename):

with open(filename) as file:

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

return maze

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

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

def bfs(maze):

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

def get\_heuristic(cell, end):

return sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2)

def a\_star(maze):

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 solve\_maze():

filename = "maze-for-u.txt"

maze = read\_maze(filename)

path1 = bfs(maze)

for place in path1:

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

result\_1 = ""

for line in maze:

result\_1 += "".join(line) + "\n"

with open("maze-for-me-done1.txt", "w") as f:

f.write(result\_1)

path\_2 = a\_star(maze)

path\_2 = path\_2[1]

for place in path\_2:

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

result\_2 = ""

for line in maze:

result\_2 += "".join(line) + "\n"

with open("maze-for-me-done2.txt", "w") as f:

f.write(result\_2)

solve\_maze()