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 Лепихина О.М.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

**Содержание**

[Введение 3](#_Toc137131106)

[Цель работы 3](#_Toc137131107)

[Задачи 3](#_Toc137131108)

[Теоретическая часть 4](#_Toc137131109)

[BFS 4](#_Toc137131110)

[A\* 7](#_Toc137131111)

[Реализация алгоритма 9](#_Toc137131112)

[Пример работы 11](#_Toc137131113)

[Заключение 12](#_Toc137131114)

[Список литературы 13](#_Toc137131115)

[Приложение 1 14](#_Toc137131116)

[Листинг программы 14](#_Toc137131117)

# 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.

# Теоретическая часть

## BFS

**BFS, или Breadth First Search** — алгоритм обхода графа в ширину. Граф — это структура из «вершин» и «ребер», соединяющих между собой вершины. По ребрам можно передвигаться от одной вершине к другой, и BFS делает это поуровнево: сначала проходит по всем ближайшим от начальной точки вершинам, потом спускается глубже.

Выглядит это так: алгоритм начинает в заранее выбранной вершине и сначала «посещает» и отмечает всех соседей этой вершины. Потом он переходит к соседям посещенных вершин, затем — дальше по тому же принципу. Из-за характера распространения, похожего на волну, алгоритм еще называют волновым. BFS — один из двух популярных алгоритмов обхода. Второй называется DFS и подразумевает обход в глубину: сначала алгоритм проходит по ребрам «вглубь» графа.

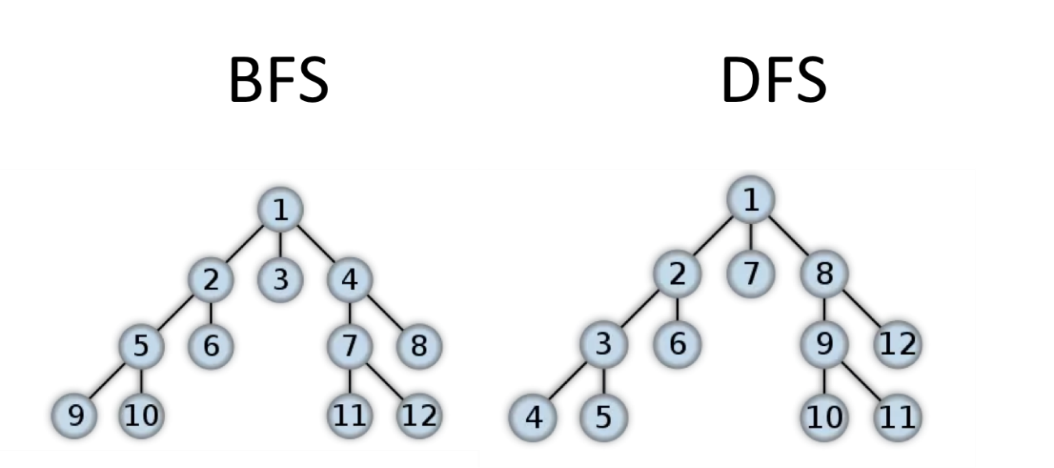


Рисунок 1. Наглядный пример работы алгоритма поиск в ширину

**Для чего нужен BFS**

* Для решения задач поиска оптимального пути. Классической задачей считается автоматизированный поиск выхода из лабиринта.
* 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.
* 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.
* Для оптимизации памяти при обходе графа в некоторых ситуациях, например для некоторых специфических структур.
* Для работы с информацией в определенных структурах данных, таких как деревья. Их тоже можно обходить с помощью алгоритма BFS, потому что они подвид графов.

**Особенности BFS**

* Константное количество действий для каждого ребра или вершины. Это важно при расчете сложности алгоритма — при выборе оптимального метода решения той или иной задачи.
* Отсутствие проблемы «бесконечного цикла»: алгоритм не попадет в него ни при каких условиях благодаря особенностям работы.
* Высокая точность и надежная архитектура, которая позволяет полагаться на этот алгоритм в решении различных задач.
* Возможность работать и с ориентированными, и с неориентированными графами.
* Полнота алгоритма — он найдет решение, то есть кратчайший путь, и завершится на любом конечном графе. Если граф бесконечный, решение найдется только в том случае, если конечен какой-либо из его путей.
* Возможность находить кратчайший путь в графе, если все ребра одинаковой длины.

**Как работает алгоритм BFS**

Алгоритм простой и интуитивно понятный. Он проходит по вершинам графа, пока в том не останется непосещенных вершин, и рассчитывает самый короткий путь до целевой вершины. Чтобы показать его работу нагляднее, представим алгоритм пошагово.

**Начало работы.** В качестве начальной можно выбрать любую вершину. На момент начала работы алгоритма все вершины помечены как непосещенные — их называют «белыми». Первое, что делает алгоритм, — помечает начальную вершину как посещенную (также используют термины «развернутая» или «серая»). Если она и есть целевая, на этом алгоритм завершается. Но чаще всего это не так.

**Поиск соседей.** Алгоритм проверяет, какие соседи есть у начальной вершины. Они добавляются в «очередь действий» в том порядке, в каком алгоритм их нашел, и тоже помечаются как «серые». Это продолжается, пока у начальной вершины не останется «белых» соседей.

**Переход на следующую вершину.** Когда алгоритм проходит по всем соседям начальной вершины, он помечает ее как полностью обойденную. Такие вершины еще называют «черными»: алгоритм к ним не возвращается. Затем он переходит к одной из «серых» вершин — соседей начальной. Алгоритм выбирает первую вершину в очереди. Далее действия повторяются: «соседи» вершины, кроме «черной», заносятся в очередь.

Когда и эта вершина будет пройдена, переход повторится по тому же принципу — первая вершина в очереди. В этом случае ею будет второй сосед начальной вершины — мы помним, что их добавляли в очередь первыми. И только когда соседи начальной вершины в очереди закончатся, алгоритм пойдет по следующему «уровню» вершин. Так и достигается обход в ширину.

**Конец алгоритма.** Если очередь оказалась пустой, это значит, что «белых» и «серых» вершин больше не осталось. Алгоритм завершится. Если при этом целевая вершина не будет достигнута, это значит, что доступа к ней из начальной точки нет.

Если целевая вершина достигается раньше, чем алгоритм пройдет по всему графу, это также может означать его завершение. Алгоритм остановится, потому что задача окажется выполнена: самый короткий путь к целевой вершине будет найден.

## A\*

(произносится «А звезда» или «А стар», от англ. *A star*) — в информатике и математике, алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Порядок обхода вершин определяется **эвристической функцией** «расстояние + стоимость» (обычно обозначаемой как *f(x)*). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (*x*) из начальной (обычно обозначается как *g(x)* и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как *h(x)*).

Функция *h(x)* должна быть **допустимой эвристической оценкой**, то есть не должна переоценивать расстояния к целевой вершине. Например, для задачи маршрутизации *h(x)* может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.

Этот алгоритм был впервые описан в 1968 году Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем. Это, по сути, было расширение алгоритма Дейкстры, созданного в 1959 году. Новый алгоритм достигал более высокой производительности (по времени) с помощью эвристики. В их работе он упоминается как «алгоритм A». Но так как он вычисляет лучший маршрут для заданной эвристики, он был назван A\*.

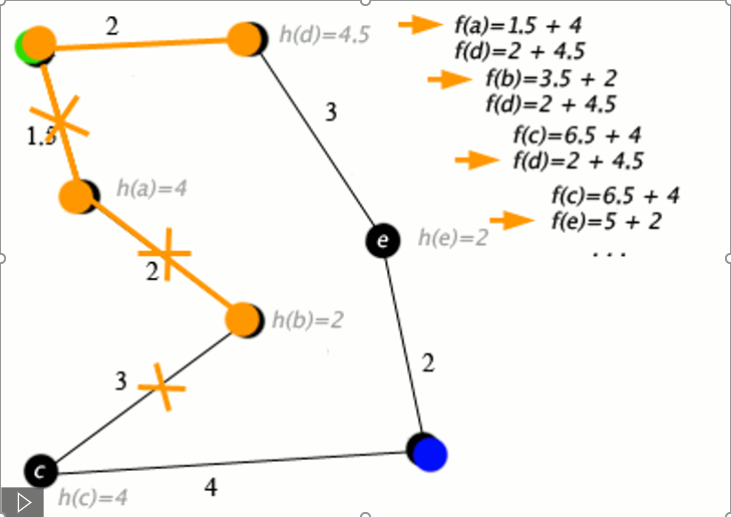


Рисунок 2. Пример алгоритма A \* в действии, где узлами являются города, соединенные дорогами, а h (x) - расстояние по прямой до целевой точки (Клавиша: зеленый: старт; синий: цель; оранжевый: посетил)

# Реализация алгоритма

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

**get\_neighbors** принимает матрицу лабиринта и координаты клетки в виде кортежа (row, col) и возвращает список соседних клеток, которые можно достичь из данной клетки. Функция проверяет, что соседние клетки находятся внутри границ лабиринта и не являются стенами (обозначенными символом «#»).

Поиск в ширину

**find\_path** в начале задаются точки старта и конца поиска, затем создаётся очередьqueue, 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.

А\*

**get\_heuristic** функция, которая принимает координаты двух клеток в виде кортежей (cell и end) и возвращает евклидово расстояние между ними. Она используется в алгоритме A\* для оценки расстояния от текущей клетки до конечной клетки.

**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.

# Пример работы

Из начала лабиринта идут точки до места, где рандомно образовался ключ, после нахождения ключа по лабиринту идут запятые до выхода из лабиринта.



Рисунок 1. Точками обозначается путь по лабиринту до A\*



Рисунок 2. Запятыми обозначается путь по лабиринту от A\* до выхода

# Заключение

В рамках выполнения данной работы были изучены теоретические основы алгоритмов обхода графа в ширину и А\*, структурного программирования.

В ходе проделанной работы был 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. Консольное приложение и получившиеся маршруты были сохранены в файл.

Структура программы состоит из модулей для работы с лабиринтом, а также модулей для реализации алгоритмов обхода в ширину и А\*.

Таким образом, цель курсовой работы достигнута. Результатом является программа, способная находить маршрут в лабиринте с помощью алгоритмов обхода графа в ширину и А\*, с сохранением полученных маршрутов в файле.

# Список литературы

1. Blog.skillfactory [электронный ресурс] - <https://blog.skillfactory.ru/glossary/bfs/> (дата обращения: 25.05.2023)
2. wikipedia [электронный ресурс]- <https://ru.wikipedia.org/wiki/A>\* (дата обращения: 25.05.2023)
3. Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser. «Structures and Algorithms in Java. Sixth Edition» - United States of America: John Wiley & Sons, Inc., 2014 г. – 720 с.
4. Алексеев В.Е., Таланов В.А. «Графы. Модели вычислений. Структуры данных» - Нижний Новгород: Издательство Нижегородского госуниверситета, 2004. – 291 с.
5. Easy A \* (звезда) Поиск пути // digitrain.ru / [Электронный ресурс]. URL: <https://digitrain.ru/articles/337034/> (Дата обращения: 25.05.2023)
6. Алгоритм поиска A \* // Wiki URL: https://ru.abcdef.wiki/wiki/A\*\_search\_algorithm (дата обращения: 08.06.2023).
7. How to Implement Breadth-First Search in Python // Python in Wonderland URL: https://pythoninwonderland.wordpress.com/tag/ai-search/ (дата обращения: 08.06.2023).

# 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  
maze = read\_maze("maze-for-u.txt")  
#создает ключ в рандомном месте  
height = len(maze)  
width = len(maze[0])  
passages = []  
for i in range(height):  
 for j in range(width):  
 if maze[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)  
 print(maze)  
 #Создание текстового документа, который рисует путь точками от входа до ключа и запятыми от ключа до выхода  
 path1 = find\_path(maze)  
 print(path1)  
 path2 = find\_path\_a\_star(maze)  
 print(path2)  
 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("maze-for-me-done.txt", "w") as f:  
 f.write(result2)  
main()