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 Дзиневский С.О

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

**Оглавление**

[Введение 3](#_Toc135785456)

[Задачи 3](#_Toc135785457)

[1. Теоретическая часть 4](#_Toc135785458)

[1.2 Алгоритм A\* 5](#_Toc135785459)

[2. Реализация алгоритма 7](#_Toc135785460)

[Пример работы 11](#_Toc135785461)

[Заключение 12](#_Toc135785462)

[Список литературы 13](#_Toc135785463)

[Листинг программы 14](#_Toc135785464)

# **Введение**

Обход графа — это переход от одной его вершины к другой в поисках свойств связей этих вершин. Связи (линии, соединяющие вершины) называются направлениями, путями, гранями или ребрами графа. Вершины графа также именуются узлами.  
  
Граф – это замечательная абстракция, к которой можно свести много различных задач и решить их через алгоритмы на графах. К графам сводятся не только задачи транспортной логистики. В виде графа можно представить, например, компьютерную сеть, сеть социальных контактов, да и вообще различные взаимоотношения между людьми, а также многие другие задачи, которые можно решить программированием.

Одним из алгоритмов является Breadth-First Search (поиск в ширину)

**Цель работы:** реализовать алгоритмы обхода графа: Breadth-First Search (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**

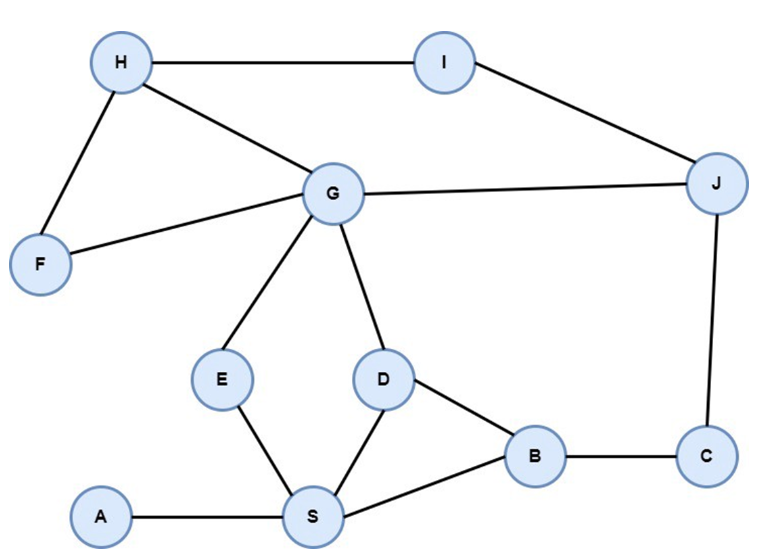
Поиск в ширину (BFS, Breadth-First Search)

[Алгоритм](http://e-maxx.ru/algo/bfs) 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.

Алгоритмы поиска применяются для решения таких задач, которые можно смоделировать как графы. Каждый узел графа – это экземпляр задачи. Каждый поисковый алгоритм начинается с узла (исходный экземпляр – состояние) и наращивает вслед за этим узлом новые (то есть, новые экземпляры задачи), решая задачу допустимыми способами. Этот процесс останавливается, как только алгоритм находит решение (успех – конечное состояние) или не может создать ни одного нового узла (провал). Среди самых популярных алгоритмов поиска – поиск в глубину (DFS), поиск в ширину (BFS), жадный алгоритм, поиск по критерию стоимости (UCS), A\*-поиск, т.д. В этой статье речь пойдет о поиске в ширину.

Поиск в ширину – это «слепой» алгоритм. Он называется «слепым», так как не учитывает стоимости перехода между вершинами графа. Алгоритм начинает работу с корневого узла (представляющего собой исходное состояние задачи) и исследует все узлы на рассматриваемом уровне, а только после этого переходит к узлам следующего уровня. Если алгоритм находит решение, то он возвращается и прекращает поиск, в противном случае наращивает от узла новое ребро и продолжает поиск. Алгоритм поиска в ширину является «полным» - это означает, что он всегда возвращает решение, если оно существует.  Точнее, алгоритм возвращает то решение, которое ближе всего к корню. Поэтому в задачах, где переход от узла к любому его дочернему узлу стоит единицу, алгоритм BFS возвращает наилучшее решение. Кроме того, чтобы исследовать узлы уровень за уровнем, он использует структуру данных под названием [очередь](https://python.plainenglish.io/queue-data-strucure-theory-and-python-implementation-e58f3582c390), так что новые узлы добавляются в хвост очереди, а старые узлы удаляются из головы очереди

Так, мы определили задачу, теперь нужно смоделировать ее в виде графа. Чаще всего для этого создается вершина на каждую комнату и ребро на каждую дверь в лабиринте. После такого моделирования получается показанный ниже граф – в нем 11 вершин и 15 ребер.



(рис.1 пример реализации)

* 1. **Алгоритм A\***

Впервые описан в 1968 году Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем. Данный [алгоритм](http://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_A*) является расширением алгоритма Дейкстры, ускорение работы достигается за счет эвристики — при рассмотрении каждой отдельной вершины переход делается в ту соседнюю вершину, предположительный путь из которой до искомой вершины самый короткий. При этом существует множество различных методов подсчета длины предполагаемого пути из вершины. Результатом работы также будет кратчайший путь.

[Алгоритм A\*](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0_A*) — распространенный алгоритм поиска пути от одной вершины к другой с наименьшей суммой весов ребер в графе. Или, другими словами, кратчайший путь из точки A в точку B. Он часто используется для программирования ИИ в играх. В [этой статье](https://shanee.io/blog/2015/10/22/generic-astar-for-games/) вы узнаете, как реализовать А\* на C++ для игр в наиболее общем виде.

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.

1. **Реализация алгоритма**

Реализация алгоритма поиска в ширину в данной программе:

1. Создаем очередь и добавляем в нее начальную точку.

2. Создаем множество посещенных вершин и добавляем в него начальную точку.

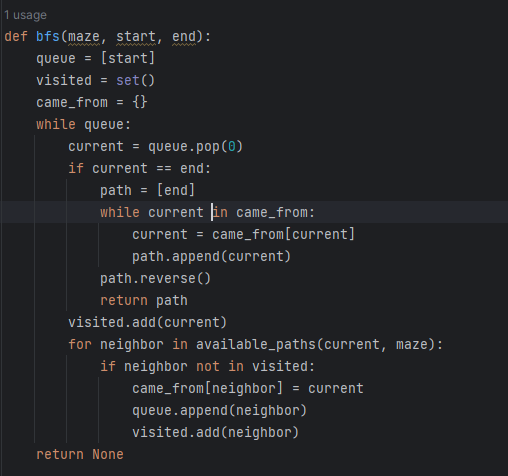
3. Создаем словарь came\_from, где ключами являются вершины, а значениями - вершины, из которых мы пришли в данную вершину. Добавляем в словарь начальную точку со значением None.

4. Пока очередь не пуста, извлекаем первую вершину из очереди.

5. Если текущая вершина равна конечной точке, то находим путь до нее, используя словарь came\_from.

6. Для каждой соседней вершины, которая еще не была посещена, добавляем ее в очередь и в множество посещенных, а также записываем в словарь came\_from текущую вершину как "предшествующую" для этой соседней вершины.

7. Если путь до конечной точки не найден, возвращаем None.



(рис.2 пример работы из программы)

Реализация алгоритма A\* в данной программе:

1. Создаем кучу heap и добавляем в нее начальную точку с f\_score = 0.

2. Создаем множество посещенных вершин и добавляем в него начальную точку.

3. Создаем словарь came\_from, где ключами являются вершины, а значениями - вершины, из которых мы пришли в данную вершину. Добавляем в словарь начальную точку со значением None.

4. Создаем словарь g\_score, где ключами являются вершины, а значениями - длины путей от начальной точки до данной вершины. Добавляем в словарь начальную точку со значением 0.

5. Создаем словарь f\_score, где ключами являются вершины, а значениями - оценки расстояния от начальной точки до конечной точки через данную вершину. Вычисляем значение f\_score для начальной точки и добавляем ее в словарь.

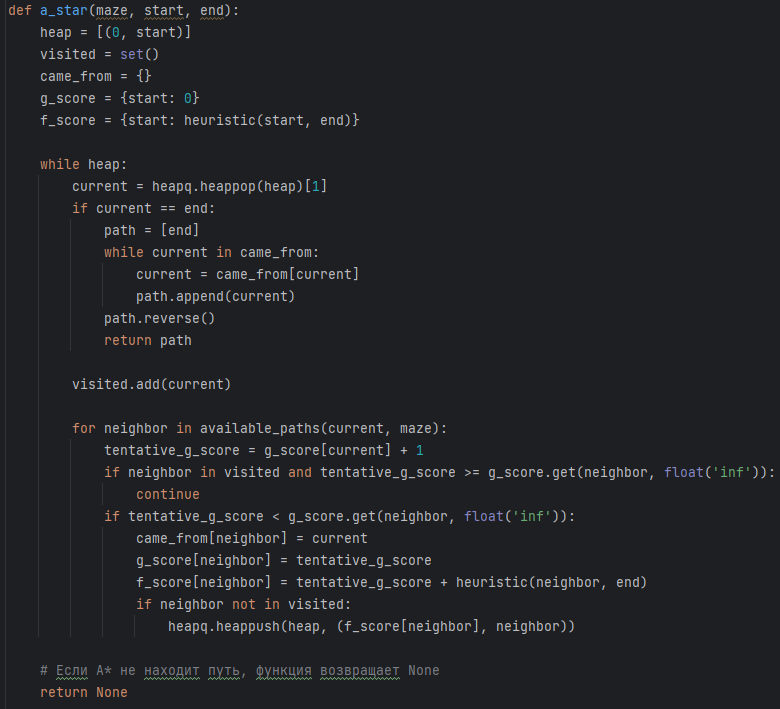
6. Пока куча heap не пуста, извлекаем из нее первую вершину с наименьшей оценкой f\_score.

7. Если текущая вершина равна конечной точке, то находим путь до нее, используя словарь came\_from.

8. Добавляем текущую вершину в множество посещенных.

9. Для каждой соседней вершины, которая еще не была посещена, вычисляем ее временную оценку g\_score через текущую вершину и ее оценку расстояния f\_score через текущую вершину до конечной точки, и добавляем ее в кучу heap.

10. Если путь до конечной точки не найден, возвращаем None.



(рис.3 пример работы из программы)

# **Пример работы**

**Поиск в ширину**

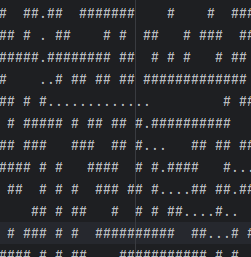
****

Рис. 4 Построенный точками путь от начала до конца лабиринта.

**Алгоритм А\***

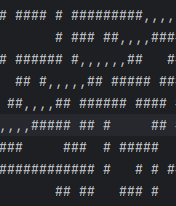


Рис. 5 Построенный путь точками от начала до конца лабиринта A\*

# **Заключение**

В рамках выполнения задачи были разработан программный код для решения задачи прохождения маршрута в лабиринте. Для этого были использованы алгоритмы поиска в ширину и A\*. Первый алгоритм позволяет найти кратчайший путь от начальной точки до конечной, а второй - находит оптимальный маршрут с учетом эвристической функции. В результате работы был создан код для прохождения маршрута от начальной координаты аватара до ключа с помощью алгоритма поиска в ширину, а затем - от ключа до ближайшего выхода с помощью A\*.

# **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: 24.05.2023).
2. Алексеев В.Е. Таланов В.А. Графы Модели вычислений Структуры данных. / Издательство Нижегородского госуниверситета. – Нижний Новгород: 2004 – 4 c. – Текст: непосредственный. (Дата обращения: 24.05.2023).
3. Рассел, С., Норвиг, П. (2009). Artificial Intelligence: A Modern Approach (3-е изд.). Москва: ДМК Пресс. (Дата обращения: 24.05.2023).
4. IT форум URL: <https://habr.com/en/companies/piter/articles/679020/>(дата обращения:30.05.2023)
5. Форум для программирования URL: <https://tproger.ru/articles/pathfindings/> (дата обращения:30.05.2023)
6. веб-сервис для хостинга IT-проектов URL: <https://lisiynos.github.io/s1/graph_alg_py.html> (дата обращения:30.05.2023)
7. Система вопросов и ответов о программировании <URL:https://ru.stackoverflow.com/> (дата обращения:30.05.2023)

# **Листинг программы**

import heapq  
  
def available\_paths(coordsXY, maze):  
 LenMazeY = len(maze[0])  
 LenMazeX = len(maze)  
 coordsX = coordsXY[0]  
 coordsY = coordsXY[1]  
 possibleWays = []  
  
 if (coordsX - 1) >= 0 and maze[coordsX - 1][coordsY] == " ": #Север  
 coord\_for\_append = (coordsX - 1, coordsY)  
 possibleWays.append(coord\_for\_append)  
  
 if (coordsY + 1) < LenMazeY and maze[coordsX][coordsY + 1] == " ": #Восток  
 coord\_for\_append = (coordsX, coordsY + 1)  
 possibleWays.append(coord\_for\_append)  
  
 if (coordsX + 1) < LenMazeX and maze[coordsX + 1][coordsY] == " ": #Юг  
 coord\_for\_append = (coordsX + 1, coordsY)  
 possibleWays.append(coord\_for\_append)  
  
 if (coordsY - 1) >= 0 and maze[coordsX][coordsY - 1] == " ": #Запад  
 coord\_for\_append = (coordsX, coordsY - 1)  
 possibleWays.append(coord\_for\_append)  
 return possibleWays  
  
  
def bfs(maze, start, end):  
 queue = [start]  
 visited = set()  
 came\_from = {}  
 while queue:  
 current = queue.pop(0)  
 if current == end:  
 path = [end]  
 while current in came\_from:  
 current = came\_from[current]  
 path.append(current)  
 path.reverse()  
 return path  
 visited.add(current)  
 for neighbor in available\_paths(current, maze):  
 if neighbor not in visited:  
 came\_from[neighbor] = current  
 queue.append(neighbor)  
 visited.add(neighbor)  
 return None  
  
  
def a\_star(maze, start, end):  
 heap = [(0, start)]  
 visited = set()  
 came\_from = {}  
 g\_score = {start: 0}  
 f\_score = {start: heuristic(start, end)}  
  
 while heap:  
 current = heapq.heappop(heap)[1]  
 if current == end:  
 path = [end]  
 while current in came\_from:  
 current = came\_from[current]  
 path.append(current)  
 path.reverse()  
 return path  
  
 visited.add(current)  
  
 for neighbor in available\_paths(current, maze):  
 tentative\_g\_score = g\_score[current] + 1  
 if neighbor in visited and tentative\_g\_score >= g\_score.get(neighbor, float('inf')):  
 continue  
 if tentative\_g\_score < g\_score.get(neighbor, float('inf')):  
 came\_from[neighbor] = current  
 g\_score[neighbor] = tentative\_g\_score  
 f\_score[neighbor] = tentative\_g\_score + heuristic(neighbor, end)  
 if neighbor not in visited:  
 heapq.heappush(heap, (f\_score[neighbor], neighbor))  
  
 # Если A\* не находит путь, функция возвращает None  
 return None  
  
  
def heuristic(current, end):  
 return abs(current[0] - end[0]) + abs(current[1] - end[1])  
  
with open('maze-for-u.txt', 'r') as f:  
 maze = [list(line.strip()) for line in f.readlines()]  
  
  
for Y in range(len(maze[0])):  
 if maze[0][Y] == " ":  
 start = (0, Y)  
 break  
  
for Y in range(len(maze[0])):  
 if maze[len(maze) - 1][Y] == " ":  
 end = (len(maze) - 1, Y)  
 break  
  
  
for i in range(len(maze)):  
 for j in range(len(maze[0])):  
 if maze[i][j] == "\*":  
 key = (i, j)  
 maze[i][j] = " "  
 break  
  
  
pathToKey = bfs(maze, start, key)  
pathToExit = a\_star(maze, key, end)  
  
#от точки-ключа до выхода "."  
for coords in pathToKey:  
 x, y = coords  
 maze[x][y] = "."  
  
#от точки-ключа до выхода ","  
for coords in pathToExit:  
 x, y = coords  
 maze[x][y] = ","  
  
  
# Записываем измененный лабиринт в файл  
with open('maze-for-me-done.txt', 'w') as f:  
 for line in maze:  
 f.write("".join(line) + "\n")