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-34 Офицеров Е.Н

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc133492807)

[Задачи 3](#_Toc133492808)

[1.Теоретическая часть 4](#_Toc133492809)

[Алгоритм поиска в ширину (ВFS) 4](#_Toc133492810)

[Алгоритм A\* 6](#_Toc133492811)

[2. Реализация алгоритма 8](#_Toc133492812)

[Как реализовать поиска в ширину 8](#_Toc133492813)

[Как реализовать А\* 8](#_Toc133492814)

[Пример работы 9](#_Toc133492815)

[Заключение 11](#_Toc133492816)

[Список литературы 12](#_Toc133492817)

[Приложение 1 13](#_Toc133492818)

[Листинг программы 13](#_Toc133492819)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании. 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.  
  
Основным алгоритмом обхода графа является поиск в ширину (Breadth-First Search, BFS).

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

# Задачи

* Изучить алгоритмы построения маршрута в графе;
* Выделить особенности реализации алгоритмов поиск в ширину и А\*;
* Подготовить исходные данные: лабиринт, координаты точек для посещения при обходе;
* Реализовать алгоритмы;
* Сохранить результаты обходов лабиринта и получившиеся маршруты в отдельный файл.

# 1.Теоретическая часть

## Алгоритм поиска в ширину (ВFS)

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.

**Кто пользуется BFS**

* Дата-сайентисты, которые работают с информацией и ее распространением, часто взаимодействуют с теорией графов.
* Разработчики, имеющие дело с определенными видами задач: поиск оптимального маршрута, программирование передвижения «умных» машин, разработка интеллектуальных систем и другие.
* Математики и другие ученые, которые работают с теорией графов как с фундаментальным научным знанием или в контексте решения практических задач.
* Инженеры-электроники: конкретно алгоритм BFS используется при трассировке печатных плат.
* Технические специалисты, работающие в телекоммуникационных системах. Там тоже активно применяется теория графов и в частности BFS.
* Сетевые инженеры, так как теория графов активно используется в сетевых технологиях. BFS, например, применяют при обходе P2P-сетей, или пиринговых сетей, а на них основаны многие сетевые протоколы. В частности, пиринговую сеть реализует BitTorrent.

**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. Если длины ребер разные, BFS найдет путь с минимальным количеством ребер, но он не обязательно будет самым коротким.

## Алгоритм A\*

Алгоритм «A Star» — это один из проверенных опытом алгоритмов, который используют для того, чтобы найти кратчайший путь между 2 вершинами графа, у которых положительный вес ребер. Данный алгоритм был описан еще в 1968 году П. Хартом, Н. Нильсоном и Б. Рафаэлем.

Алгоритм «A Star» всегда находит кратчайший путь, потому что применяет в своих расчетах вспомогательную функцию, которая всегда направляет поиск в нужном направлении, сокращая его длительность. Такая вспомогательная функция называется «эвристика», поэтому алгоритм «A Star» относят к категории «эвристических алгоритмов поиска».

Алгоритм «A Star» характеризуется тремя важными свойствами:

* оптимальностью — это означает, что алгоритм гарантирует получение лучшего из возможных результатов;
* полнотой — это означает, что алгоритм «A Star» всегда найдет решение, если оно существует;
* эффективностью — на сегодняшний день нет других алгоритмов, которые смогут найти кратчайший путь быстрее, чем «A Star», применяя эвристическую функцию.

Алгоритм поиска «A Star» несет в себе следующую идею: изначально он всегда посещает вершины, которые, скорее всего, ведут по кратчайшему пути к цели. Такие вершины он определяет по формуле:

F(x) = G(x) + H(x) , где:

* F(x) — это функция для вершины; чем меньше функция, тем «ближе» вершина стоит в очереди для посещения; данная функция оценивает минимальную стоимость перехода от вершины к вершине;
* G(x) — это стоимость пути от первоначальной вершины и до любой другой;
* H(x) — это эвристический показатель стоимости пути от вершины «х» и до конечной вершины.

Вычисляя F(x), за «х» берется соседняя вершина с уже посещенной. Поэтому получается, что алгоритм вычисляет стоимость всего пути у всех соседних вершин и останавливает выбор на той, у которой стоимость минимальна.

# 2. Реализация алгоритма

## Как реализовать поиска в ширину

Граф обычно представляется как массив, очередь или другая структура данных. Ее элементы — вершины, и в них хранятся сведения о других вершинах, с которыми они соединены. Иногда там напрямую есть ссылки на другие вершины — конкретная реализация зависит от языка программирования и выбранной архитектуры.

Алгоритм BFS, реализованный программно, начинает с заданного элемента этой структуры данных — это аналогично начальной вершине. Он отмечает ее посещенной — например, записывает в элемент какое-то значение-метку. Затем он смотрит, на какие элементы ссылается начальный, и отмечает посещенными уже их. Когда все ссылки на другие элементы в начальной вершине заканчиваются, граф помечает ее как полностью обойденную — для этого используется другое значение-метка. Оно показывает алгоритму, что больше возвращаться в эту вершину нет смысла, — так он не «застрянет» в одних и тех же точках.

После этого алгоритм переходит по ссылке к первому найденному «соседу» этой вершины и повторяет те же действия. Это продолжается, пока все вершины не окажутся помеченными как полностью обойденные.

## Как реализовать А\*

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.

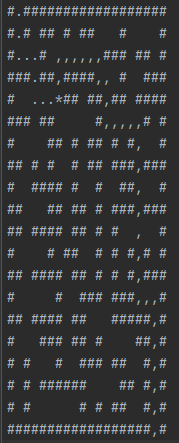
Когда одна из вершин становится полностью исследованной, она попадает в категорию «исследованные вершины», а все ее соседи попадают в категорию «известные вершины» и становятся годными для исследования. На каждой уже исследованной вершине устанавливается указатель до той уже исследованной вершины, к 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, однако его также часто применяют в играх.

# 

# Пример работы



# Заключение

В ходе выполнения задания были изучены алгоритмы построения маршрута в графе, в частности алгоритмы поиска в ширину и А\*. В результате выполнения задания результаты обходов лабиринта (от начала до ключа с помощью алгоритма поиска в ширину и от ключа до выхода с помощью алгоритма А\*) были сохранены а отдельный файл, в котором точка ключа будет указана как '\*', а сам маршрут построен точками к ключу и запятыми от него к выходу.

# Список литературы

1. BFS [Электронный ресурс]. URL: [https://blog.skillfactory.ru/glossary/bfs/](https://blog.skillfactory.ru/glossary/bfs/%20) (Дата обращения: 27.04.2023).
2. Алгоритм A Star [Электронный ресурс]. URL: <https://codernet.ru/articles/web/poiskovyij_algoritm_a_star_chto_eto_i_kak_effektivno_ego_ispolzovat/> (Дата обращения: 27.04.2023).
3. Обход графа [Электронный ресурс]. URL: <https://habr.com/ru/articles/504374/> (Дата обращения: 27.04.2023).

# Приложение 1

## Листинг программы

from queue import PriorityQueue

from queue import Queue

# Функция для импорта и преобразования исходного файла

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")

# Функция определения начальной точки

def start\_points(maze):

for Y in range(len(maze[0])):

if maze[0][Y] == " ":

start = (0, Y)

break

return start

start = start\_points(maze)

# Функция нахождения точки выхода

def end\_points(maze):

for Y in range(len(maze[0])):

if maze[len(maze) - 1][Y] == " ":

end = (len(maze) - 1, Y)

break

return end

end = end\_points(maze)

# Функция определения ключевой точки

def key\_points(maze):

for i in range(len(maze)):

for j in range(len(maze[0])):

if maze[i][j] == "\*":

key = (i, j)

break

return key

key = key\_points(maze)

def valid\_moves(maze, current\_position):

moves = []

x, y = current\_position

directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

for d in directions:

dx, dy = d

nx, ny = x + dx, y + dy

if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and maze[nx][ny] != 1:

moves.append((nx, ny))

return moves

def bfs(maze, start, end):

# Создаём очередь

queue = Queue()

queue.put(start)

visited = {start: None}

# Определяем возможные направления движения

directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# Запускаем цикл поиска пути

while not queue.empty():

current = queue.get()

# Если точка является конечной, то создаём путь до неё

if current == end:

path = []

while current is not None:

path.append(current)

current = visited[current]

return path[::-1]

# Перебираем возможные направления движения

for direction in directions:

# Вычисляем следующую точку

neighbor = (current[0] + direction[0], current[1] + direction[1])

# Проверяем, находится ли следующая точка в лабиринте

if 0 <= neighbor[0] < len(maze) and 0 <= neighbor[1] < len(maze[0]) and neighbor not in visited and maze[neighbor[0]][neighbor[1]] != '#':

# Добавляем следующую точку в очередь

queue.put(neighbor)

visited[neighbor] = current

return None

Up\_to\_the\_key = bfs(maze, start, key)

def astar(maze, start, end):

explored = set()

pq = PriorityQueue()

pq.put((0, start))

path = {}

# Начинаем алгоритм A\*

while not pq.empty():

# Извлекаем клетку из очереди с минимальным приоритетом

current\_cost, current\_node = pq.get()

# Добавляем клетку в список посещенных

explored.add(current\_node)

# Если мы нашли конечную клетку, то строим путь

if current\_node == end:

return build\_path(start, end, path)

# Перебираем соседние клетки

for child\_node in get\_neighbors(maze, current\_node):

if child\_node in explored:

continue

new\_cost = current\_cost + 1

if child\_node not in [item[1] for item in pq.queue]:

# Вычисляем эвристическую функцию

h = heuristic(child\_node, end)

# Добавляем соседнюю клетку в очередь с приоритетами

pq.put((new\_cost + h, child\_node))

path[child\_node] = current\_node

# Если клетка уже была добавлена в очередь,то обновляем путь

elif new\_cost < current\_cost(child\_node):

pq.put((new\_cost + h, child\_node))

path[child\_node] = current\_node

return None

def get\_neighbors(maze, node):

i, j = node

# Перебираем соседние клетки

neighbors = [(i+1, j), (i-1, j), (i, j+1), (i, j-1)]

valid\_neighbors = []

for neighbor in neighbors:

x, y = neighbor

if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] != '#':

valid\_neighbors.append(neighbor)

return valid\_neighbors

def heuristic(node, end):

# Манхэттенское расстояние

return abs(node[0] - end[0]) + abs(node[1] - end[1])

def build\_path(start, end, path):

# Строим путь от конечной клетки до начальной

node = end

path\_list = [end]

while node != start:

node = path[node]

path\_list.append(node)

return path\_list[::-1]

Before\_the\_exit = astar(maze,key, end)

def Signs(maze, path, symbol):

for cord in path:

x, y = cord

maze[x][y] = symbol

x, y = key

maze[x][y] = "\*"

return maze

maze\_to\_the\_key = Signs(maze, Up\_to\_the\_key, ".")

maze\_to\_exit = Signs(maze, Before\_the\_exit, ",")

def final\_file(maze, filename):

with open(filename, "w") as file:

for row in maze:

for elem in row:

file.write(str(elem))

file.write("\n")

final\_file(maze, "maze-for-me-done.txt")