# Федеральное государственное бюджетное образовательное учреждение высшего образования «Нижегородский государственный архитектурно-строительный университет» (ННГАСУ)

*Факультет инженерно-экологических систем и сооружений*

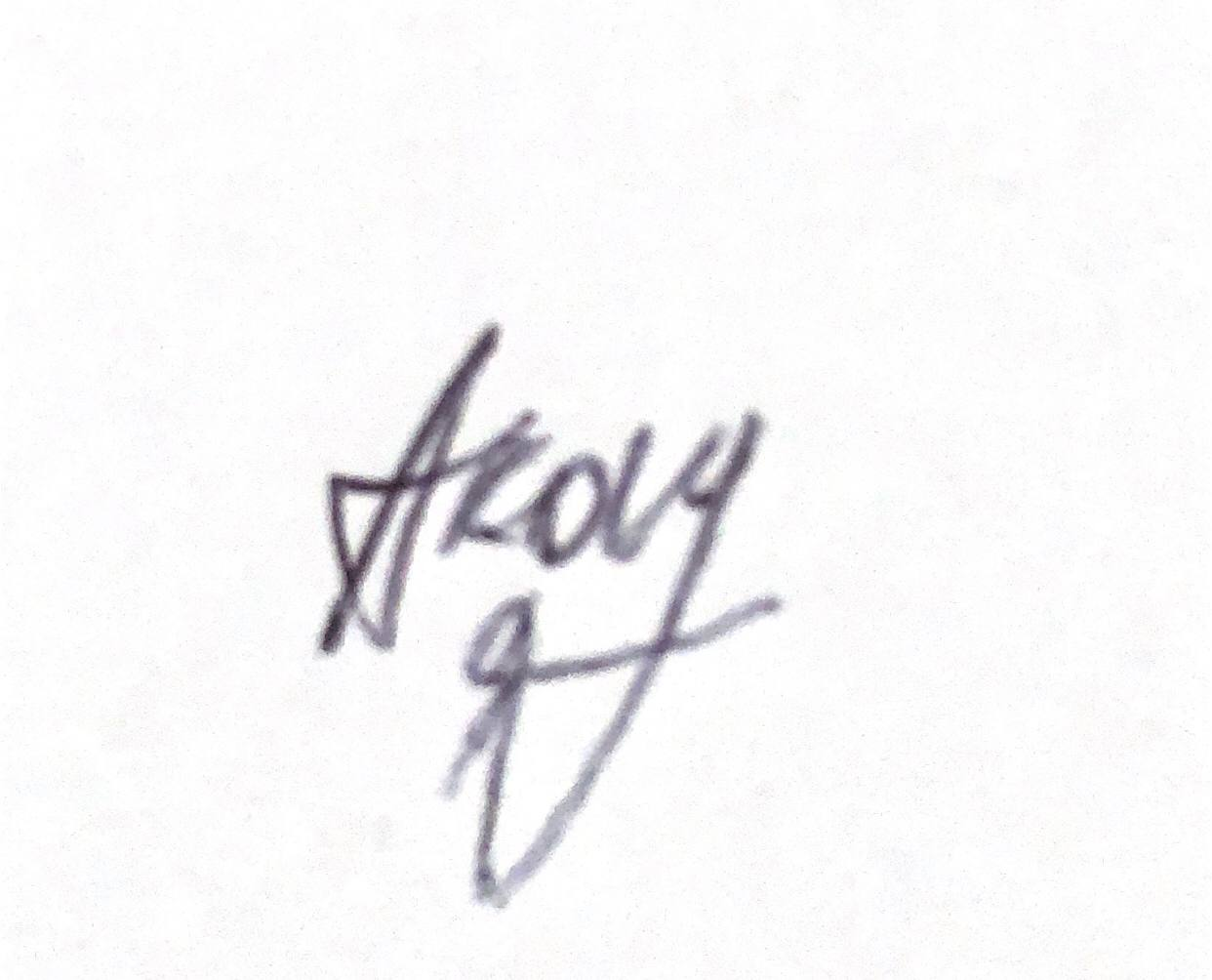
*Кафедра информационных систем и технологий*

КУРСОВАЯ РАБОТА

по дисциплине: «Язык программирования Python»

На тему: «Алгоритмы поиска пути и структурное программирование»

Выполнила студентка 1 курса гр. ИС-34 Кузнецова А.А

Подпись 

Проверил Морозов Н.С.

Дата

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc136634496)

[1.Теоретическая часть 4](#_Toc136634497)

[Алгоритм обхода графа в ширину (ВFS) 4](#_Toc136634498)

[Алгоритм A\* 6](#_Toc136634499)

[2. Реализация алгоритма 8](#_Toc136634500)

[Реализация поиска в ширину 8](#_Toc136634501)

[Реализация А\* 10](#_Toc136634502)

[Пример работы 12](#_Toc136634503)

[Заключение 13](#_Toc136634504)

[Список литературы 14](#_Toc136634505)

[Приложение 1 15](#_Toc136634506)

[Листинг программы 15](#_Toc136634507)

# 

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании. Графовые алгоритмы представляют собой последовательность шагов для обхода графа через вершины (узлы). Некоторые алгоритмы используются для поиска определенного узла или пути между двумя заданными узлами.

Основным алгоритмом обхода графа является поиск в ширину (Breadth-First Search, BFS).

Поиск в ширину — это рекурсивный алгоритм поиска всех вершин графа или дерева. Обход начинается с корневого узла и затем алгоритм исследует все соседние узлы. Затем выбирается ближайший узел и исследуются все неисследованные узлы. При использовании BFS для обхода любой узел графа можно считать корневым узлом.

**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 с заданными параметрами;
* Создать новый файл для хранения лабиринта с реализованными алгоритмами и отмеченным путём.

**1.Теоретическая часть**

**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. О том, чем они различаются, можно прочитать в статье про ориентированный граф.

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. Для поиска кратчайшего пути в таком случае будет лучше алгоритм Дейкстры.

## 

## Алгоритм A\*

Алгоритм A\* в Python или вообще в основном представляет собой задачу искусственного интеллекта, используемую для поиска пути (из точки A в точку B) и обхода графа. Этот алгоритм является гибким и может быть использован в широком диапазоне контекстов. Алгоритм поиска A\* использует эвристическую стоимость пути, стоимость начальной точки и конечную точку. Этот алгоритм был впервые опубликован Питером Хартом, Нильсом Нильссоном и Бертрамом Рафаэлем в 1968 году.

А\* - алгоритм поиска пути использует эвристику для нахождения кратчайшего пути в графе. Каждый узел имеет стоимость f(n), которая рассчитывается как f(n)=g(n) + h(n). В этой формуле g(n) - это фактическая стоимость узла с начала пути, а h(n) - его эвристическая стоимость для достижения цели. A\* является допустимым, что означает оно всегда находит решение с оптимальным значением.

Эффективность

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.

A\* в Python-это мощный и полезный алгоритм со всем потенциалом. Однако он хорош лишь настолько, насколько хороша его эвристическая функция, которая весьма изменчива, учитывая природу проблемы. Он нашел свое применение в программных системах машинного обучения и поисковой оптимизации для разработки игр.

Приоритетная очередь в Python

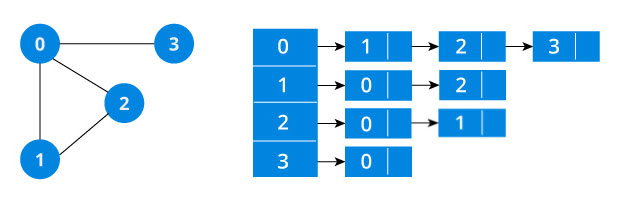
Очередь с приоритетом – это особый тип очереди в структуре данных. Как следует из названия, она сортирует элементы и удаляет их в соответствии с приоритетами. В отличие от обычной, она извлекает элемент с наивысшим приоритетом вместо следующего элемента. Приоритет отдельных элементов определяется порядком их ключей.

Priority Queue возвращает элемент в порядке приоритета. Поэтому, когда мы добавляем элемент в очередь, мы также предоставляем ему приоритет. Затем мы получаем элементы, они возвращаются в порядке приоритета. Сначала возвращаются товары с более низким приоритетом.

**2. Реализация алгоритма**

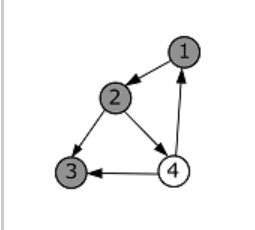
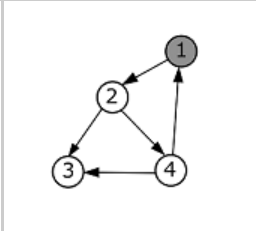
## Реализация поиска в ширину

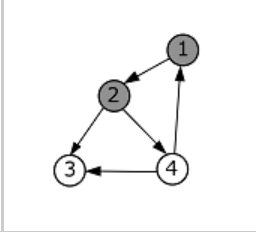
Будем рассматривать реализацию на списке смежности как самую легкую для понимания. Вкратце напомню, что список смежности - это один из способов представления графа, который выглядит примерно так:



Для каждой вершины (первый столбик) мы составляем список смежных ей вершин, то есть список вершин с которыми у данной есть общие ребра(ребро инцидентное данным вершинам).

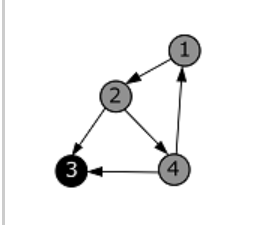
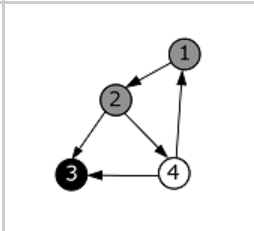
Теперь собственно к самому алгоритму, принцип его работы совпадает с его названием, данный алгоритм идет "внутрь" графа, до того момента как ему становится некуда идти, затем возвращается в предыдущую вершину и снова идет от нее до тех пор, пока есть куда идти. И так далее. Для понимания данного алгоритма нам потребуются 3 цвета, один будет обозначать, что вершину мы еще не посетили, второй что посетили и ушли, а третий, что посетили и не смогли идти дальше и начинаем возвращаться обратно.





Стартуем из любой вершины, например, из первой. Идем по списку смежности. Из 1 вершины попадаем во вторую, переходим в ее список смежности, не забываем красить 1 вершину в серый, так как мы ее посетили и ушли дальше. Из второй вершины идем в любую из списка смежности второй вершины, например в 3. Красим 2 в серый и идем в список смежности 3-й вершины. А в нем ничего нет. В таком случае мы понимаем, что дальше нам идти не куда, пора возвращаться.

Красим 3 в черный так как нам идти некуда(нет белых вершин, в которые мы могли бы пойти из 3). Возвращаемся в 2 и в ее список смежности, видим что там еще есть вершина 4, выдвигаемся туда, оттуда можно пойти только в 1, но она уже серая, то есть мы там уже были. Алгоритм начнет рекурсивно откатываться назад перекрашивая все вершины в черный, думаю принцип уже понятен.



## Реализация А\*

Алгоритм А\* — один из самых эффективных алгоритмов поиска кратчайшего пути между двумя точками графа. Он был опубликован Питером Хартом, Нильсом Нильссоном и Бертрамом Рафаэлем в 1968 году. В каком-то смысле его можно назвать расширением алгоритма Дейкстры. Однако, несмотря на это он является одним из самых часто используемых алгоритмов поиска пути.

Говоря простым языком, алгоритм А\* находит оптимальный вариант благодаря вычислению суммарной стоимости всех путей между начальной и конечной точкой. Кстати, этот способ быстрее алгоритма Дейкстры благодаря эвристической функции.

f(n) = g(n) + h(n)

f(n): общая стоимость пути

g(n): стоимость пути между текущей и начальной вершиной

h(n): эвристическая функция

# Пример работы

Для примера работы был взят небольшой лабиринт, где “ # ” - стена, “ \* ” - ключ, “ . ” – путь от входа до ключа, “,” – путь от ключа до выхода, “ ” – поля по которым можно двигаться. Результат можно посмотреть на рисунке 1.

# 

# Рисунок 1. Представление лабиринта

# 

**Заключение**

В ходе проделанной работы были изучены алгоритмы построения маршрута в графе, в частности алгоритмы поиска в ширину и А\*. В результате была реализована программа поиска пути с заданными параметрами на языке программирования Python. В результате выполнения задания была создана программа, которая при помощи двух алгоритмов пути возвращает результат пути по координатам от точки начала до ключа и от ключа до конца лабиринта. Результатом данной работы является программа, которая способна находить самый оптимальный путь, используя алгоритмы А\*.

# Список литературы

1. BFS- что это за алгоритм [Электронный ресурс]. URL: [https://blog.skillfactory.ru/glossary/bfs/](https://blog.skillfactory.ru/glossary/bfs/%20) (Дата обращения: 24.05.2023).
2. A\* search algorithm[Электронный ресурс]. URL: [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)(дата обращения: 26.05.2023).
3. Breadth-first search [Электронный ресурс]. URL: <https://en.wikipedia.org/wiki/Breadth-first_search>(дата обращения 26.05.2023)
4. Метод поиска пути в лабиринте при наличии помех [Электронный ресурс]. URL: (<https://cyberleninka.ru/>) (Дата обращения: 26.05.2023).
5. Алгоритм А\* История создания. [Электронный ресурс]. URL:[https://studfile.net/preview/3239584/page:4/#11](https://studfile.net/preview/3239584/page:4/%2311). (дата обращения: 28.05.2023)
6. Реализация алгоритма A\*. [Электронный ресурс]. URL: [https://habr.com/ru/articles/331220//.](https://habr.com/ru/articles/331220/.%20) ( дата обращения: 28.05.2023)
7. Алгоритма A\* и его реализация. [Электронный ресурс]. URL:<https://pythonist.ru/algoritm-a-star-i-ego-realizacziya-na-python/>.(дата обращения: 28.05.2023)

# Приложение 1

## Листинг программы

import heapq

from collections import deque

# 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 bfs(maze, start, end):

# инициализация очереди и множества посещенных вершин

queue = deque([(start, [])])

visited = set([start])

# продолжаем искать, пока очередь не опустеет

while queue:

# извлечь вершину из очереди

node, path = queue.popleft()

# если мы достигли цели, вернуть найденный путь

if node == end:

return path

# добавить все непосещенные соседние вершины в очередь

for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

x, y = node[0]+dx, node[1]+dy

if 0 <= x <len(maze) and 0 <= y < len(maze[0]) and maze[x][y] != '#' and (x, y) not in visited:

visited.add((x, y))

queue.append(((x, y), path + [(x, y)]))

# если мы не нашли путь до цели, возвращаем None

return None

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

def heuristic(a, b):

return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(maze, start, end):

path = []

frontier = []

heapq.heappush(frontier, (0, start))

came\_from = {}

cost\_so\_far = {}

came\_from[start] = None

cost\_so\_far[start] = 0

while frontier:

current = heapq.heappop(frontier)[1]

if current == end:

break

for next in get\_neighbors(maze, current):

new\_cost = cost\_so\_far[current] + 1

if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

cost\_so\_far[next] = new\_cost

priority = new\_cost + heuristic(end, next)

heapq.heappush(frontier, (priority, next))

came\_from[next] = current

if current != end:

return None

while current != start:

path.append(current)

current = came\_from[current]

path.append(start)

path.reverse()

return path

def get\_neighbors(maze, current):

neighbors = []

for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

x = current[0] + dx

y = current[1] + dy

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

neighbors.append((x, y))

return neighbors

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")