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 Котов Д. В.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

**Содержание**

[Введение 3](#_Toc136123275)

[Цель работы: 3](#_Toc136123276)

[Задачи: 3](#_Toc136123277)

[Основная часть 4](#_Toc136123278)

[Алгоритмы поиска пути 4](#_Toc136123279)

[Алгоритм поиска в глубину 4](#_Toc136123280)

[Алгоритм А\* 4](#_Toc136123281)

[Структурное программирование 6](#_Toc136123282)

[Реализация алгоритма 7](#_Toc136123283)

[Реализация алгоритма поиска в глубину 7](#_Toc136123284)

[Реализация алгоритма A\* 7](#_Toc136123285)

[Пример работы алгоритма 8](#_Toc136123286)

[Заключение 9](#_Toc136123287)

[Список литературы 10](#_Toc136123288)

[Приложения 11](#_Toc136123289)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании. Алгоритмы поиска пути являются одними из наиболее распространенных алгоритмов в области компьютерного зрения и машинного обучения. Поиск пути – типичная задача программирования, решаемая в самых разных ситуациях. Она известна в основном из навигационных задач и разработки игр. Но, изучив ключевые алгоритмы поиска пути, вы узнаете, что они применимы к более абстрактным задачам оптимизации и построения последовательностей.

Цель работы:реализовать алгоритмы обхода графа: Поиск в глубину и A\* для задачи поиска маршрута в лабиринте.

## Задачи:

1. Запустить скрипт gen\_lab\_origin.py, подождать его работу несколько минут;
2. Полученный файл переместить в свою папку;
3. Изучить теорию алгоритмов обхода графа: в глубину и А\*.
4. Написать программу на языке Python для поиска маршрута в лабиринте с помощью алгоритма обхода в глубину и А\*;
5. Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

# Основная часть

## Алгоритмы поиска пути

### Алгоритм поиска в глубину

Поиск в глубину – вероятно, наиболее важная ввиду многочисленности приложений стратегия обхода графа. Идея этого метода – идти вперед в неисследованную область, пока это возможно, если же вокруг все исследовано, отступить на шаг назад и искать новые возможности для продвижения вперед. Метод поиска в глубину известен под разными названиями, например, «бэктрекинг», «поиск с возвращением». Алгоритм поиска пути в глубину широко применяются в задачах компьютерного зрения. Они используются для поиска объектов на изображениях и видео, а также для распознавания образов. Главное отличие от поиска в ширину состоит в том, что при поиске в глубину в качестве активной выбирается та из открытых вершин, которая была посещена последней. Для реализации такого правила выбора наиболее удобной структурой хранения множества открытых вершин является стек: открываемые вершины складываются в стек в том порядке, в каком они открываются, а в качестве активной выбирается последняя вершина. Схематически это показано на рис.1(Приложение 1) [3, c.47-54].

Алгоритм поиска в глубину работает следующим образом:

1. Начните с того, что поместите любую вершину графа на вершину стека.
2. Возьмите верхний элемент стека и добавьте его в список “Пройденных”.
3. Создайте список смежных вершин для этой вершины. Добавьте те вершины, которых нет в списке “Пройденных”, в верх стека.
4. Необходимо повторять шаги 2 и 3, пока стек не станет пустым.

### Алгоритм А\*

Алгоритм A\* известен как эффективный метод поиска оптимального маршрута от начальной до конечной вершины с наименьшей стоимостью. Его относят к информированным алгоритмам поиска, так как для решения задач используются данные о стоимости пути и принципы [эвристики](https://ru.wikipedia.org/w/index.php?search=%D1%8D%D0%B2%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B0&title=%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F%3A%D0%9F%D0%BE%D0%B8%D1%81%D0%BA&go=%D0%9F%D0%B5%D1%80%D0%B5%D0%B9%D1%82%D0%B8). Алгоритм А\* обладает двумя ключевыми характеристиками алгоритмов такого рода: *оптимальность* и *полнота*. Если алгоритм поиска характеризуется как оптимальный, значит он гарантирует получение лучшего из возможных решений. А когда среди характеристик присутствует определение «полный», это означает, что алгоритм всегда находит решение, если таковое существует. Также в процессе работы алгоритма для верщин рассчитывается функция – f(v)=g(v)+h(v), где g(v) - наименьшая стоимость пути в v из стартовой вершины, а g(v) - эвристическое приближение стоимости пути от v до конечной цели [1].

Он просматривает все возможные маршруты, уделяя внимание тем, которые кажутся наиболее перспективными. Это отличает его от жадного алгоритма, который выбирает первый наиболее подходящий маршрут.

В то время как алгоритм A\* оптимален для «случайно» заданных графов, нет гарантии, что он сделает свою работу лучше, чем более простые, но и более информированные относительно проблемной области алгоритмы. Например, в некотором лабиринте может потребоваться сначала идти по направлению от выхода, и только потом повернуть назад. В этом случае обследование вначале тех вершин, которые расположены ближе к выходу (по прямой), будет потерей времени.

## Структурное программирование

Структурное программирование — это методология программирования, которая предполагает использование только трех основных структур управления: последовательности, ветвления и цикла. Эти структуры позволяют создавать четкие и простые программы .

Одно из главных преимуществ структурного программирования заключается в том, что оно нацелено на создание легко читаемого и сопровождаемого кода. Это достигается путем разделения программы на небольшие блоки с помощью последовательностей, ветвлений и циклов таким образом, чтобы код был понятным и структурированным для других разработчиков, которые могут участвовать в дальнейшей поддержке проекта.

Однако, структурное программирование имеет и недостатки. Некоторые задачи, которые можно решить легко с помощью конструкций GOTO, могут быть сложными или невозможными в структурном программировании. Кроме того, этот метод может стать громоздким при выполнении сложных логических условий, что затем приведет к затратам на ресурсы и удлинению времени разработки.

Несмотря на это, структурное программирование до сих пор используется и является неотъемлемой частью языков программирования, таких как C, Pascal, Ada, и многих других. Это отличный способ улучшения понимания и читабельности кода для более эффективной разработки и программирования [5].

# Реализация алгоритма

## Реализация алгоритма поиска в глубину

Алгоритм поиска в глубину (DFC – Death-First Search) используется для обхода графа или дерева. Идея заключается в том, что мы двигаемся от начальной вершины (точки, места) в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или пункта назначения (искомой вершины). Алгоритм заключается вот в чем сначала нам нужно поместить начальную вершину в стек, и если стек пуст, то завершиться поиск. Затем извлечь вершину из стека, и после этого может произойти один из двух вариантов:1) Если эта вершина является конечной, то завершить поиск, 2) Если эта вершина не была посещена, то пометить ее как посещенную и добавить все ее соседние вершины в стек. Затем повторить данные шаги (кроме помещения начальной вершины в стек).

## Реализация алгоритма A\*

Сначала используется функция get\_heuristic(), которая в дальнейшем нужна для оценки расстояния от каждой точки до конечной точки.

Затем функция find\_path\_a\_star реализует поиск пути в лабиринте с помощью алгоритма A\*.

Также в начале алгоритма присутствуют начальная и конечный точки start и end, а также приоритетная очередь queue.

Далее создается множество visited, в котором в свою очередь будут хранится посещенные точки. Это множество создается для того чтобы избежать повторных посещений.

Затем в цикле while, пока очередь не пуста, извлекается список с минимальной стоимостью из приоритетной очереди.

После этого проверяется, является ли текущая точка конечной точкой.

Далее идет еще одна проверка, 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.

И в конце, если конечная точка не может быть достигнута из начальной точки, то функция возвращает None.

# Пример работы алгоритма

1. Используем программу “gen\_lab\_origin.py” для создания файла “maze-for-u.txt” и получаем лабиринт (Приложение 2).
2. Используем программу “lab3.py” и после ее запуска появляется текстовый файл “maze-for-me-done” (я ждал почти 2 часа чтобы создался данный файл).
3. Открываем текстовый файл и видим результат (Приложение 3)

# Заключение

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

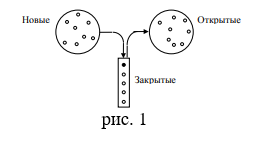
Алгоритмы являются одними из наиболее распространенных алгоритмов в области компьютерного зрения и машинного обучения. Они могут быть использованы для решения различных задач, таких как поиск кратчайшего пути между двумя вершинами, поиск пути с минимальной стоимостью и т.д. Структурное программирование, в свою очередь, является методом проектирования программного обеспечения, который может помочь упростить код и сделать его более понятным и легко поддерживаемым.

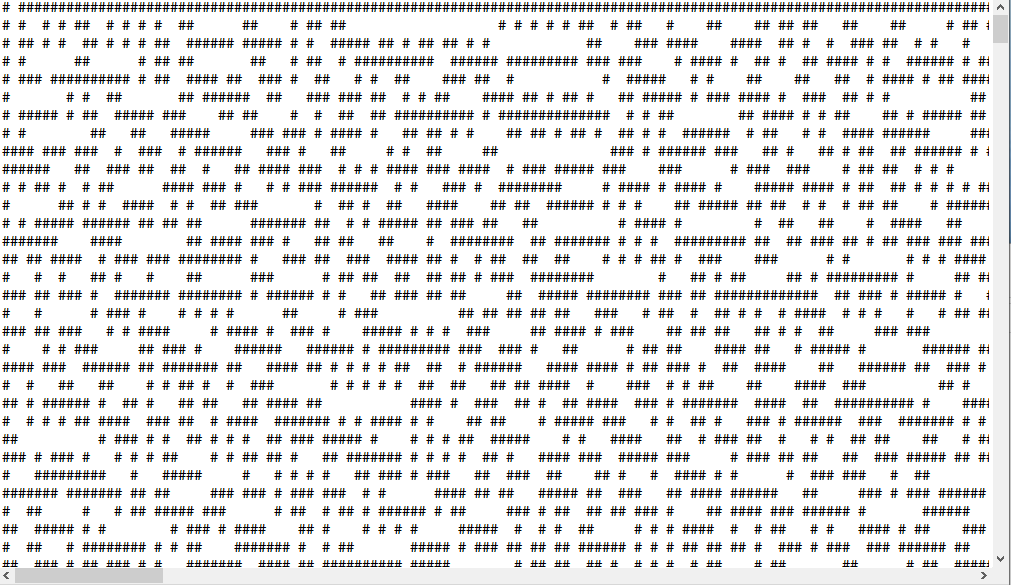
Таким образом, цель курсовой работы достигнута. Результатом является программа, способная находить маршрут в лабиринте с помощью алгоритмов обхода графа в глубину и А\*.

# Список литературы

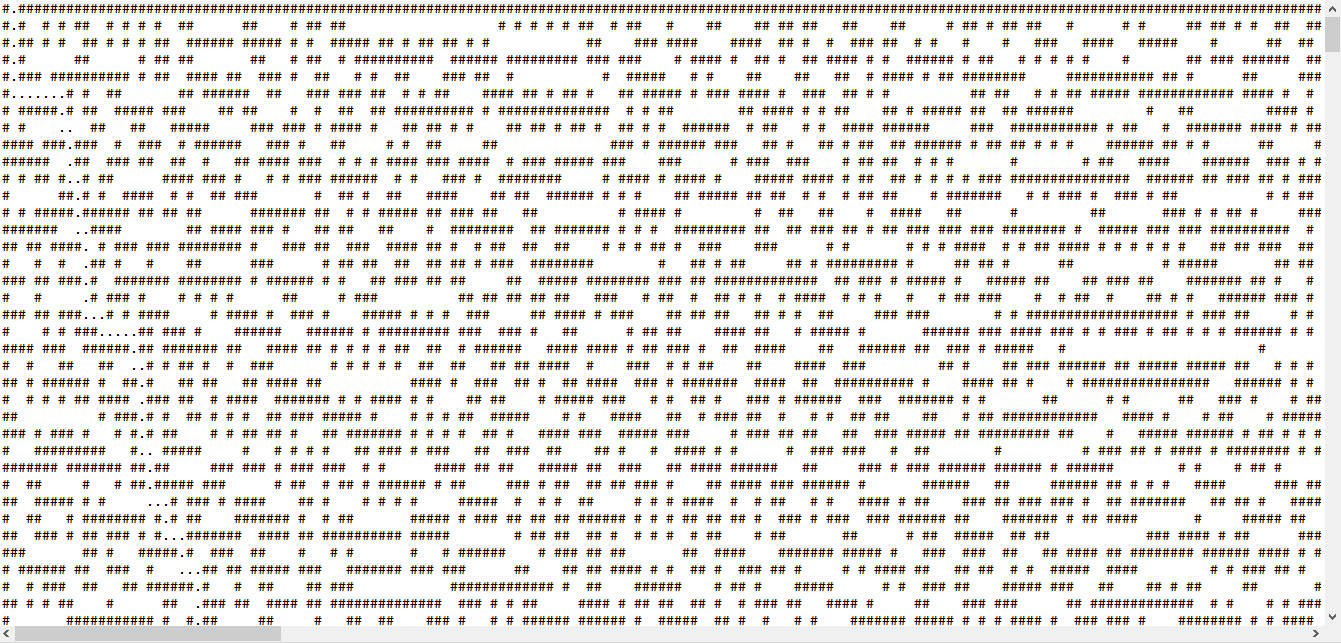
1. Алгоритм поиска A\*. Пошаговый разбор алгоритма поиска A\*// medium.com / [Электронный ресурс]. URL: <https://medium.com/nuances-of-programming/алгоритм-поиска-a-3bb59be05a79> (Дата обращения 26. 05.2023)
2. Алгоритм A\*[Электронный ресурс]. URL: [https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм\_A\*&mobileaction=toggle\_view\_mobile](https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_A*&mobileaction=toggle_view_mobile) (Дата обращения 26.05.2023)
3. Алексеев В.Е., Таланов В.А. «Графы. Модели вычислений. Структуры данных» - Нижний Новгород: Издательство Нижегородского госуниверситета, 2004. – 291 с.
4. 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: 27.05.2023).
5. Лекция 10. Структурное программирование, предпрограммная подготовка задачи// StudFiles / [Электронный ресурс]. URL: <https://studfile.net/preview/4290050/> (Дата обращения: 26.05.2023)
6. Обход графа[Электронный ресурс]. URL: <https://habr.com/en/articles/504374/> (Дата обращения 26.05.2023)
7. Поиск в глубину [Электронный ресурс]. URL: <https://ru.wikipedia.org/wiki/Поиск_в_глубину> ( 25.05.2023)

# Приложения



**Приложение 1**.

**Приложение 2. Первоначальный вид лабиринта**

****

**Приложение 3. Готовый лабиринт с путем, состоящим из точек и запятых**

**Приложение 4. 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

height = len(read\_maze("maze-for-u.txt"))

width = len(read\_maze("maze-for-u.txt")[0])

passages = []

for i in range(height):

for j in range(width):

if read\_maze("maze-for-u.txt")[i][j] == " ":

passages.append((i, j))

random\_key = random.choice(passages)

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

# Поиск в глубину

def find\_path(maze):

start = (0, 1)

key = random\_key

stack = [(start, [start])]

visited = set()

while stack:

current, path = stack.pop()

if current == key:

return path

visited.add(current)

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

if neighbor not in visited:

stack.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)

#Создание текстового документа, который рисует путь точками от входа до ключа и запятыми от ключа до выхода

path1 = find\_path(maze)

path2 = find\_path\_a\_star(maze)

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()