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: «Реализация алгоритмов поиска пути в лабиринте»

Выполнил студент 1 курса гр. ИС-34 Лобанов А.С.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 4](#_Toc135835454)

[Задачи 4](#_Toc135835455)

[Глава 1. Теоретическая часть 5](#_Toc135835456)

[1.1 Алгоритм поиска в ширину 5](#_Toc135835457)

[1.2 Алгоритм А\* 5](#_Toc135835461)

[2. Реализация алгоритма 6](#_Toc135835462)

[2.1 Реализация алгоритма поиска в ширину 6](#_Toc135835463)

[2.2 Реализация алгоритма А\* 6](#_Toc135835471)

[Пример работы 8](#_Toc135835472)

[Заключение 10](#_Toc135835473)

[Список литературы 11](#_Toc135835474)

[Приложение 1 12](#_Toc135835475)

[Листинг программы 12](#_Toc135835476)

# Введение

Данная курсовая работа направлена на создание алгоритма поиска пути в лабиринте. Поиск пути в лабиринте является одной из основных задач в области робототехники и искусственного интеллекта, а также имеет широкое применение в автоматизации производства, игровой индустрии и грузоперевозках. В работе будут рассмотрены два алгоритма - BFS и А\*, проведен анализ их преимуществ и недостатков, а также оценена производительность и сложность. Реализация алгоритмов будет выполнена на языке программирования Python.

Главной целью работы является создание алгоритмов обхода графа - BFS и А\* для решения задачи поиска маршрута в лабиринте.

# Задачи

• Изучение алгоритмов построения маршрута в графе.

• Выделение особенностей реализации, которые необходимы в конкретной задаче поиска маршрута.

• Подготовка исходных данных, включающих в себя лабиринт и координаты точек для посещения при обходе.

• Реализация алгоритмов с заданными параметрами.

• Сохранение результатов обходов лабиринта и получившихся маршрутов в файл.

# Глава 1. Теоретическая часть

**1.1 Графы**

Для начала, стоит определить, что такое граф. Граф - это математическая абстракция реальной системы, объекты которой обладают парными связями. Граф как математический объект представляет собой совокупность двух множеств: множество вершин и множество ребер. Элементом множества ребер является пара элементов множества вершин [5].

В данной работе автор рассматривает граф, имеющий двумерный вид, то есть граф, построенный по 2 координатам. Такой граф можно встретить во многих учебниках, в программе Excel и в других местах. Для работы программы граф был преобразован из текстового файла, содержащего 3 условных символа: « » - свободное пространство, по которому можно перемещаться в пределах самого лабиринта, «#» - стена, через которую нельзя проходить и «\*» - ключ, условное место, находящееся в свободном пространстве, через которое обязательно надо пройти.

Сам код принимает данный граф как двумерный массив, заполненный символами « », «#» и «\*».

**1.2 Алгоритмы построения маршрута в графе, используемые в разработке**

В ходе работы над кодом автор использовал 2 алгоритма построения маршрута в графе: поиск в ширину и A\*. Поиск в ширину использовался для поиска пути до ключа, особой точки, через которую нужно пройти перед тем, как попасть на выход. Поиск до выхода производился с помощью алгоритма A\*.

Обход или поиск является одной из фундаментальных операций, выполняемых на графах. Поиск в ширину начинается с определенной вершины, затем исследуются все её соседи на данной глубине, после чего происходит переход к вершинам следующего уровня. В графах, в отличие от деревьев, могут быть циклы - пути, в которых первая и последняя вершины совпадают. Поэтому необходимо отслеживать посещенные алгоритмом вершины. При реализации алгоритма поиска в ширину используется структура данных "очередь".

A\* пошагово просматривает все пути, ведущие от начальной вершины к конечной, пока не найдет минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые "кажутся" ведущими к цели. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до нее путь. Составляющая g(x) - это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме [8].

A\* является оптимальным алгоритмом, так как он использует эвристику, которая позволяет выбирать вершины, близкие к цели, в первую очередь. Временная сложность A\* зависит от качества эвристики, а также от того, как быстро можно обновить значения функций f и g. В худшем случае временная сложность A\* составляет O(b^d), где b - размер ветвления, а d - глубина.

## 1.3 Алгоритм поиска в ширину

## BFS (Breadth-First Search) - это алгоритм поиска в ширину в графе, который используется для обхода графа или нахождения кратчайшего пути между двумя вершинами.

## Описание алгоритма:

## Алгоритм BFS гарантирует нахождение кратчайшего пути в невзвешенном графе, то есть графе, где все ребра имеют одинаковый вес. Он также может быть использован для нахождения кратчайшего пути во взвешенном графе, если все веса ребер равны.

## 1.4 Алгоритм А\*

Алгоритм A\* - это эвристический алгоритм поиска кратчайшего пути в графе с весами ребер. Он отличается от алгоритма Дейкстры тем, что использует эвристическую оценку расстояния от текущей вершины до целевой вершины для выбора следующей вершины в пути, что позволяет достигать большей производительности.

Одним из главных преимуществ алгоритма A\* является его оптимальность при использовании допустимой эвристической функции. Это означает, что эвристическая функция не должна переоценивать действительную стоимость пути. Алгоритм A\* широко используется в робототехнике, видеоиграх, планировании маршрутов и других областях, где требуется нахождение оптимальных путей в графах.

**1.5 Особенности реализации**

Для реализации кода и написания текста автор использовал язык программирования Python 3.11 и среду разработки PyCharm. Python был выбран из-за его простоты и удобства в освоении, а также из-за наличия необходимых функций. Среда разработки PyCharm была выбрана, потому что автор считает ее интуитивно понятной и удобной в использовании.

В коде был реализован аватар, у которого есть текущее местоположение, соседи и пройденный путь. Для алгоритма поиска в ширину были необходимы только координаты аватара, ключа и соседних точек. Для алгоритма А\* были необходимы координаты ключа, выхода и прямой путь от ключа до выхода, по которому будет построен путь до выхода.

Одна из особенностей реализации заключается в том, что ребра графа имеют единичную длину. В этом случае алгоритм А\* может не показать себя наилучшим образом, но в данной задаче этого достаточно. Также был создан файл, в котором был построен путь в виде точек до ключа и запятых от ключа до выхода. В некоторых местах путь повторяется, и в таком случае он обозначается запятыми.

В целом, реализация кода была достаточно простой. Однако, чтобы все работало корректно, было необходимо учитывать множество мелких деталей, таких как правильная работа с координатами и создание нужного формата файла. В результате был получен код, который эффективно находит кратчайший путь в графе и создает файл с путем в нужном формате.

# 2. Реализация алгоритма

## 2.1 Реализация алгоритма поиска в ширину

## Создание очереди queue, содержащей только стартовую точку, и списка visited, содержащего только стартовую точку.

## Запуск цикла while, который будет выполняться до тех пор, пока queue не станет пустым.

## Извлечение первой точки из очереди.

## Проверка, является ли текущая точка конечной. Если да, то происходит восстановление пути и его возврат.

## Получение списка соседних точек для текущей точки, проверка их наличия в пределах лабиринта и на проходимость.

## Если соседняя точка не была посещена ранее, то она добавляется в очередь и в список visited.

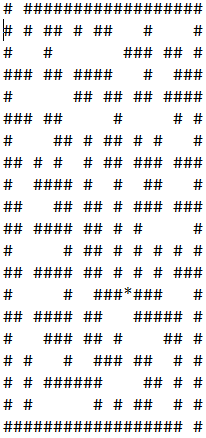
## Если путь до конечной точки не был найден, то возвращается сообщение "Тайлер дерден?".

## 2.2 Реализация алгоритма А\*

1. Создание списка points\_set, содержащего только стартовую точку.
2. Запуск цикла while, который будет выполняться до тех пор, пока points\_set не станет пустым.
3. Выбор точки с наименьшей оценкой f(x) = g(x) + h(x) из points\_set.
4. Проверка, является ли текущая точка конечной. Если да, то происходит восстановление пути и его возврат.
5. Получение списка соседних точек для текущей точки, проверка их наличия в пределах лабиринта и на проходимость.
6. Если соседняя точка уже находится в points\_set, то сравнивается длина пути до этой точки через текущую точку и ее предыдущую длину пути. Если новый путь короче, то обновляется длина пути и предыдущая точка для соседней точки.
7. Если соседняя точка не находится в points\_set, то она добавляется в список.
8. Если путь до конечной точки не был найден, то возвращается сообщение "Тайлер дерден?".

# Пример работы

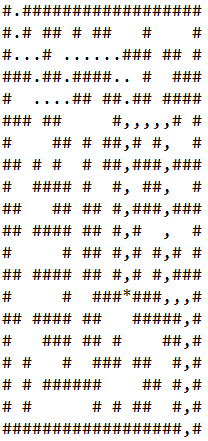
Изначальный файл:



Ключ, до которого нужно добраться аватару

Конечный файл:

Точками отмечен путь до ключа, маршрут начало-ключ



Запятыми отмечен путь от ключа до выхода, так же перекрывает «путь из точек», т.к. этот маршрут был проложен позже маршрута начало-ключ

Ключ

# Заключение

Была создана программа на языке программирования Python 3.11, предназначенная для поиска пути в лабиринте. Для достижения данной цели были изучены и применены алгоритмы BFS и A\*.

Программа осуществляет работу с текстовыми файлами и позволяет строить путь в лабиринте. В процессе ее разработки был изучен и применен ряд основных принципов работы указанных алгоритмов.

# Список литературы

1. A\* search algorithm[Электронный ресурс]: Википедия. Свободная энциклопедия. – Режим доступа: [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)(дата обращения: 25.04.2023).
2. "Поиск в ширину" / [Электронный ресурс] // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/Поиск\_в\_ширину (дата обращения: 27.04.2023).
3. Алгоритмы: построение и анализ: учебник / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — 2-е изд. — М.: [«Вильямс»](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D0%BB%D1%8C%D1%8F%D0%BC%D1%81_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)&action=edit&redlink=1" \o "Вильямс (издательство) (страница отсутствует)), 2006. — 1296 с.
4. Дольников, В. Л. Основные алгоритмы на графах: текст лекций / В. Л. Дольников, О. П. Якимова. - Ярославль: ЯрГУ, 2011. – 80 с. – URL: <http://www.lib.uniyar.ac.ru/edocs/iuni/20110210.pdf>(дата обращения 26.04.2023) – Текст: электронный
5. Стаут Б. Алгоритмы поиска пути: Статья. пер: Каменский М., 2000. URL: <http://www.oim.ru/reader.asp?nomer=366>(дата обращения: 29.04.2023)
6. Агапов И., «Обход графа: поиск в глубину и поиск в ширину простыми словами на примере JavaScript» [Электронный ресурс]. URL: <https://habr.com/ru/articles/504374/> (Дата обращения: 11.04.2023).
7. Станковец А.В. АЛГОРИТМЫ НА ГРАФАХ / Станковец А.В. – Текст: электронный // M[ODERN SCIENCE](https://www.elibrary.ru/contents.asp?id=44150018). — 2020. — №10-2. — С.532-536. — EDN: [pdmxpx](https://www.elibrary.ru/pdmxpx" \o "https://www.elibrary.ru/pdmxpx)

# Приложение 1

## Листинг программы

**from** collections **import** deque

**class** Point\_bfs:

**def** \_\_init\_\_(self, y, x, prev\_point=None):

self.x = x

self.y = y

self.prev\_point = prev\_point

#-----

**def** bfs(maze, start, end):

**if** maze[start.y][start.x] == '#' **or** maze[end.y][end.x] == '#':

**return** "Тайлер дерден?"

**if** (start.y, start.x) == (end.y, end.x):

**return** [(start.y, start.x)]

*# очередь точек типа*

queue = deque([start])

*# бывалые...*

visited = [start]

*# пока здесь кто нибудь есть*

**while** queue:

*# буквально украл*

cur = queue.popleft()

*# ееее*

**if** (cur.y, cur.x) == (end.y, end.x):

path = []

**while** cur != None:

path.append((cur.y, cur.x))

cur = cur.prev\_point

*# попой вперед как и в прошлый раз*

**return** path[::-1]

*# рамки приличия*

pos = [Point\_bfs(y, x, cur) **for** y, x **in** [(cur.y+1,cur.x),(cur.y-1,cur.x),(cur.y,cur.x+1),(cur.y,cur.x-1)] \

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

**for** new\_point **in** pos:

*# Впервые здесь?*

**if** **not** any((point.y,point.x) == (new\_point.y,new\_point.x) **for** point **in** visited):

*# добавляем клетку в очередь и отмечаем ее как бывалую*

queue.append(new\_point)

visited.append(new\_point)

*# Опять?*

**return** "Тайлер дерден?"

#------------------------------------------------------

**class** Point\_A\_star:

**def** \_\_init\_\_(self, y, x, end\_point=None, prev\_point=None):

self.x = x

self.y = y

self.g = prev\_point.g + 1 **if** prev\_point != None **else** 0

*# От начальной точки до текущей точки*

self.h = abs(end\_point.x - self.x) + abs(end\_point.y - self.y) **if** end\_point != None **else** 0

*# ерет.. эврик.. еврейская функция короче*

self.prev\_point = prev\_point

*# Папа*

#-----

**def** A\_star(maze, start, end):

**if** maze[start.y][start.x] == '#' **or** maze[end.y][end.x] == '#':

**return** "Тайлер дерден?"

**if** (start.y, start.x) == (end.y, end.x):

**return** [(start.y, start.x)]

*# ну типа бывалые*

points\_set = [start]

*# пока здесь кто нибудь есть*

**while** points\_set:

cur = min(points\_set, key=**lambda** x: x.g+x.h)

*# Выбор точки с наименьшей оценкой f(x) = g(x) + h(x)*

points\_set.remove(cur)

*# Пакета*

**if** (cur.y, cur.x) == (end.y, end.x):

*# Если текущая точка является конечной точкой*

path = []

**while** cur != None:

*# Восстановление пути*

path.append((cur.y, cur.x))

cur = cur.prev\_point

*# т.к. попа впереди то*

**return** path[::-1]

pos = [Point\_A\_star(y, x, end, cur) **for** y, x **in** [(cur.y+1,cur.x),(cur.y-1,cur.x),(cur.y,cur.x+1),(cur.y,cur.x-1)] \

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

*# Соседние точки в пределах рамок приличия*

**for** new\_point **in** pos:

old\_point = next((point **for** point **in** points\_set **if** (point.y,point.x) == (new\_point.y,new\_point.x)), None)

*# Пытемся понять была ли эта точка до new\_point*

**if** old\_point != None:

*# Если была, то надо бы объективно понять, действительно ли раньше было лучше*

**if** new\_point.g < old\_point.g:

*# Сравнение путей от начала говорит о том что раньше лучше не было (((*

old\_point.g = new\_point.g

old\_point.prev\_point = new\_point.prev\_point

*# Добавим свежести*

**else**:

points\_set.append(new\_point)

*# Или же это не хорошо забытое старое, а что то новое*

*# Тайлер дерден?*

**return** "Тайлер дерден?"

#------------------------------------------------------

**def** add\_path(maze, path1, path2):

**if** path1 != "Тайлер дерден?" **and** path2 != "Тайлер дерден?":

**for** y,x **in** path1:

maze[y][x]='.'

**for** y,x **in** path2:

maze[y][x]=','

**with** open('done\_maze.txt', 'w') **as** f:

**for** line **in** maze:

f.write("".join(line) + "\n")

**else**: **print**('Это Тайлер')

**with** open('maze.txt', 'r') **as** f:

maze = [list(line.strip()) **for** line **in** f.readlines()]

*# start, key, end*

**for** x **in** range(len(maze[0])):

**if** maze[0][x]==' ': start = (0,x)

**for** x **in** range(len(maze[-1])):

**if** maze[-1][x]==' ': end = (len(maze)-1, x)

y = len(maze)//2; x = len(maze[0])//3

**while** maze[y][x]!=' ': x+=1; y+=1

*# отметить звездочкой обязательно*

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

key = (y,x)

path1 = bfs(maze, Point\_bfs(\*start), Point\_bfs(\*key))[:-1]

path2 = A\_star(maze, Point\_A\_star(\*key), Point\_A\_star(\*end))[1:]

add\_path(maze, path1, path2)

**print**("finally")