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 Данилова М.Д.

Проверил Морозов Н.С.

Нижний Новгород

2023 г.

**Содержание**

[**Введение** 3](#_Toc136775959)

[**Задачи** 3](#_Toc136775960)

[**1.Теоретическая часть** 3](#_Toc136775961)

[**1.1 Поиск пути по ширине** 3](#_Toc136775962)

[**1.2 Алгоритм А\*** 4](#_Toc136775963)

[**2. Реализация алгоритма** 5](#_Toc136775964)

[**2.1 Реализация поиска по ширине** 5](#_Toc136775965)

[**2.2 Реализация алгоритма А\*** 5](#_Toc136775966)

[**Пример работы** 7](#_Toc136775967)

[**Заключение** 8](#_Toc136775968)

[**Список литературы** 9](#_Toc136775969)

[**Приложения** 10](#_Toc136775970)

# **Введение**

Цель курсовой работы – написание программы поиска пути в лабиринте. Поиск пути в лабиринте является одной из главных задач в области робототехники и находит широкое применение в других различных областях.

В данной работе мы рассмотрим два алгоритма поиска пути в лабиринте: поиск по ширине и алгоритм A. Реализация алгоритма поиска пути в лабиринте будет выполнена на языке программирования Python.

**Цель работы**: реализовать алгоритмы обхода графа: по ширине и А\* для поиска маршрута в лабиринте.

# **Задачи**

* Изучить алгоритмы построения маршрута в графе;
* Выделить особенности реализации, необходимые в конкретной задаче поиска маршрута;
* 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**

## **1.1 Поиск пути по ширине**

Поиск в ширину (англ. breadth-first search) — один из основных алгоритмов на графах, позволяющий находить все кратчайшие пути от заданной вершины и решать многие другие задачи.

Поиск в ширину также называют обходом — так же, как поиск в глубину и все другие обходы, он посещает все вершины графа по одному разу, только в другом порядке: по увеличению расстояния до начальной вершины.

В терминах логического программирования вычисление цели *j* приводит к полному обходу в глубину конкретного дерева поиска цели *j*, в котором выбирается самая левая цель. По существу здесь декларирован некоторый порядок извлечения целей. Действительно, говоря о поиске в глубину, мы имеем в виду тот порядок, в котором рассматриваются альтернативы в пространстве состояний. В соответствии с этим порядком запоминается одна альтернативная вершина и связанные с ней потомки. Весь проход в глубину образует путь, который, возможно, не достигнет цели, поэтому сработает механизм возврата для поиска альтернативы, вновь ведущей вглубь.

## 

Рис. 1 Пример работы алгоритма

## **1.2 Алгоритм А\***

Алгоритм А \*— один из самых популярных методов решения задач на поиск кратчайшего маршрута. Его относят к информированным алгоритмам поиска, так как для решения задач используются данные о стоимости пути и принципы эвристики. 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**.

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все [информированные алгоритмы поиска](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От [жадного алгоритма](https://ru.wikipedia.org/wiki/%D0%96%D0%B0%D0%B4%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Составляющая *g(x)* — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение *f(x)*, после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в [очереди с приоритетом](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_%D1%81_%D0%BF%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BE%D0%BC_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Приоритет пути определяется по значению *f(x) = g(x) + h(x)*. Алгоритм продолжает свою работу до тех пор, пока значение *f(x)* целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

Чем меньше эвристика *h(x)*, тем больше приоритет, поэтому для реализации очереди можно использовать [сортирующие деревья](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B5%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE).

Множество просмотренных вершин хранится в closed, а требующие рассмотрения пути — в очереди с приоритетом open. Приоритет пути вычисляется с помощью функции *f(x)* внутри реализации очереди с приоритетом.

# **2. Реализация алгоритма**

## **2.1 Реализация поиска по ширине**

Поиск в ширину работает путём последовательного просмотра отдельных уровней [графа](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), начиная с узла-источника�.

Рассмотрим все [рёбра](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B1%D1%80%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2))(�,�), выходящие из узла�. Если очередной узел� является целевым узлом, то поиск прекращается; в противном случае узел� добавляется в [очередь](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). После того, как будут проверены все рёбра, выходящие из узла�, из очереди извлекается следующий узел�, процесс повторяется.

1. Поместить узел, с которого начинается поиск, в изначально пустую очередь.
2. Извлечь из начала очереди узел � и пометить его как развёрнутый.
   * Если узел � является целевым узлом, то завершить поиск с результатом «успех».
   * В противном случае, в конец очереди добавляются все преемники узла �, которые ещё не развёрнуты и не находятся в очереди.
3. Если очередь пуста, то все узлы [связного графа](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D0%B3%D1%80%D0%B0%D1%84) были просмотрены, следовательно, целевой узел недостижим из начального; завершить поиск с результатом «неудача».
4. Вернуться к п. 2.

Примечание: деление вершин на развёрнутые и не развёрнутые необходимо для произвольного графа (так как в нём могут быть циклы). Для [дерева](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)) эта операция не нужна, так как каждая вершина будет выбрана один-единственный раз.

## **2.2 Реализация алгоритма А\***

1. Создается 2 списка вершин- ожидающие рассмотрения и уже рассмотренные. В ожидающие добавляется точка старта, список рассмотренных пока пуст.

2. Для каждой точки рассчитывается F=G+H.

G- расстояние от старта до точки,

H- примерное расстояние от точки до цели

Каждая точка хранит ссылку на точку, из которой в нее пришли.

3. Из списка точек на рассмотрение выбирается точка с наименьшим F. Обозначаем ее X.

4. Если X- цель, то маршрут найден.

5. Переносим X из списка ожидающих рассмотрения в список уже рассмотренных.

6. Для каждой из точек, соседних для X(обозначим эту точку Y).

7. Если Y уже находится в рассмотренных- пропускаем ее.

8. Если Y еще нет в списке на ожидание- добавляем ее туда, запомнив ссылку на X и рассчитав Y.G(X.G + расстояние от X до Y) и Y.H.

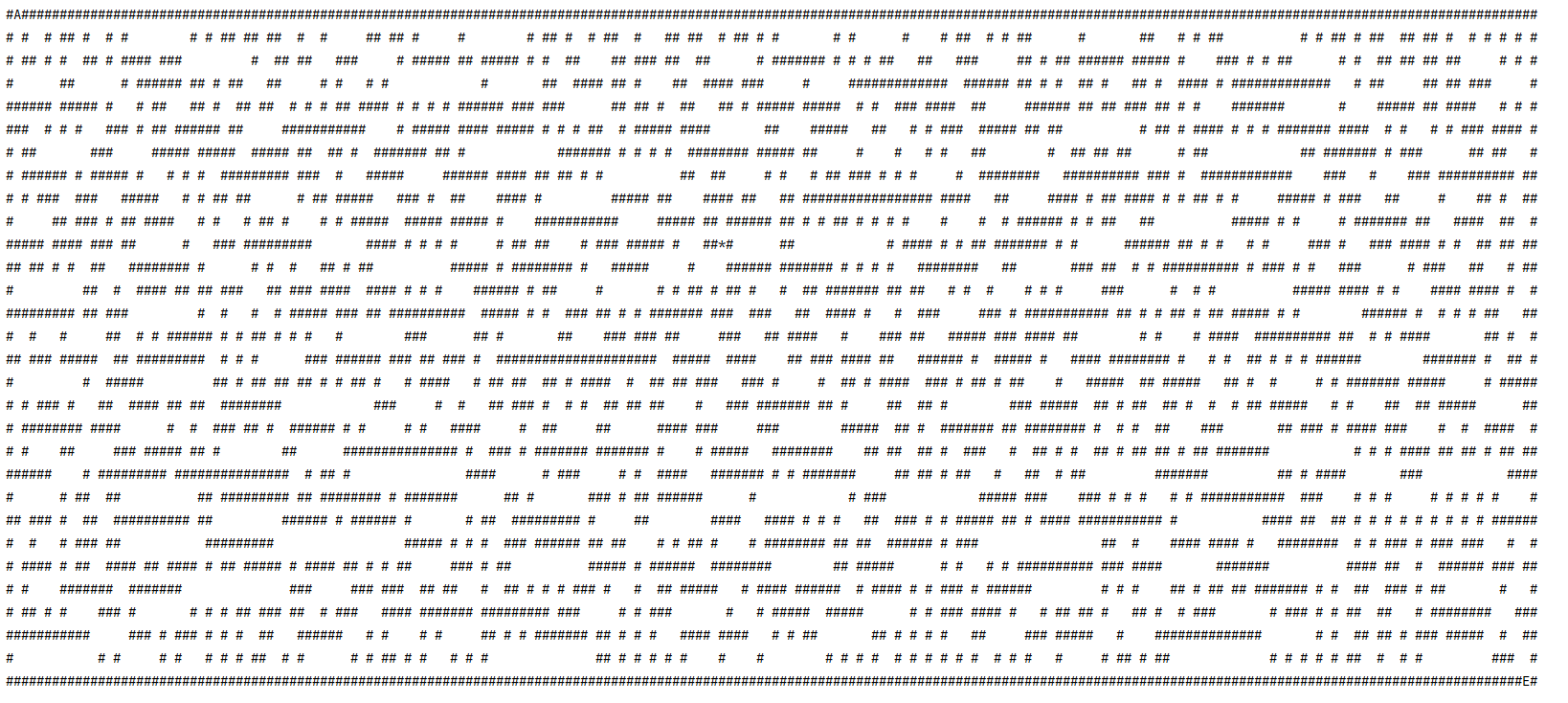
9. Если Y в списке на рассмотрение- проверяем, если X.G + расстояние от X до Y<Y.G, значит, мы пришли в точку Y более коротким путем, заменяя Y.G на X.G+ расстояние от X до Y, а точку, из которой пришли в Y на X.

10. Если список точек на рассмотрение пуст, а до цели так и не дошли- значит маршрута не существует.[2]

# 

# **Пример работы**

Ниже приложены скриншоты результатов работы программного кода.



По достижению сокровища «\*» маршрут до выхода начинает строится с помощью алгоритма А\* и помечается «,». (рис. 2)

Рис.2 Исходный лабиринт

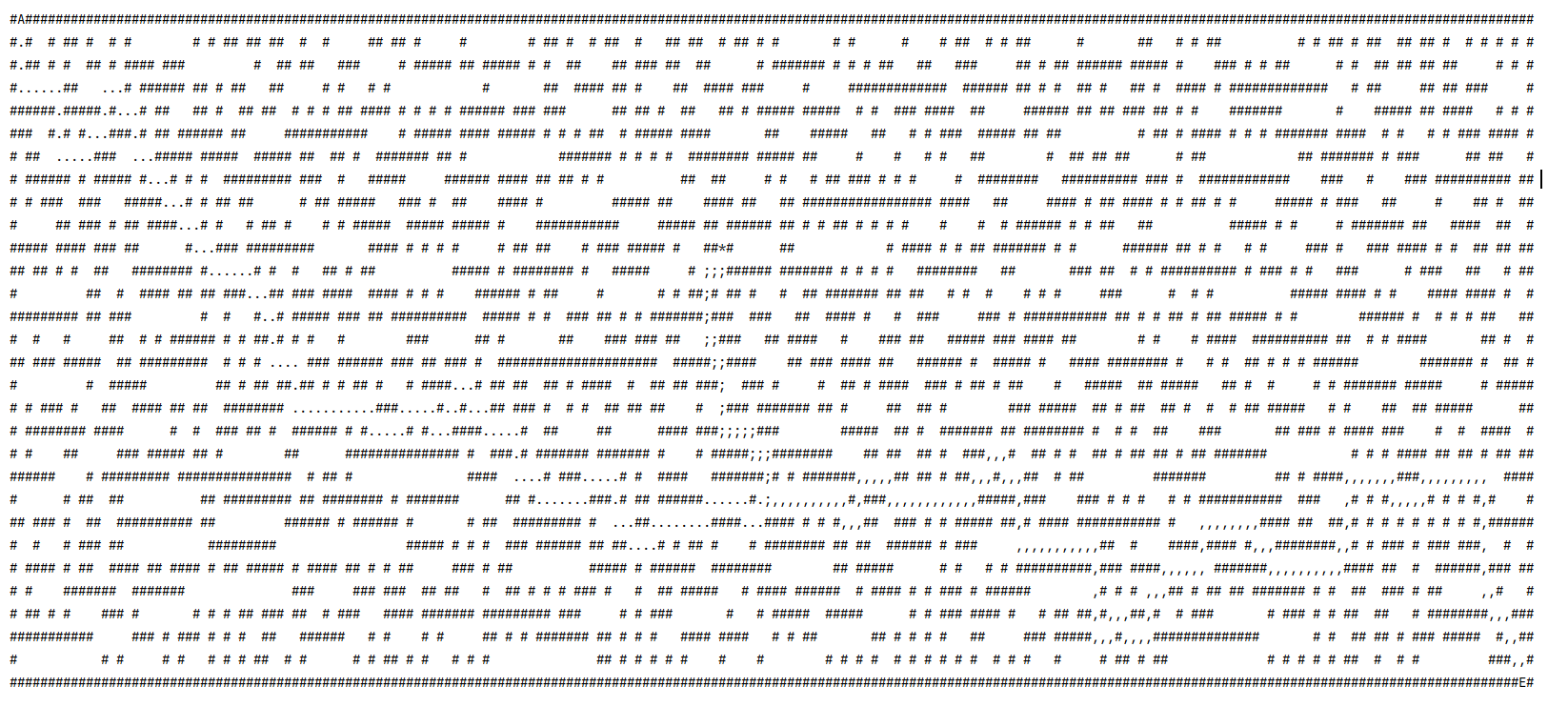


Рис.3 Результат работы

# 

# **Заключение**

В ходе проделанной работы были освоены алгоритмы построения маршрута, в частности алгоритмы поиска по ширине и А\*. Был подготовлен лабиринт, реализованы алгоритмы с заданными параметрами. Для построения маршрута в лабиринте, использовался алгоритм поиска по ширине от начала лабиринта до ключа «\*», и алгоритмом А\* был построен путь от ключа «\*» до выхода.

Алгоритм поиска в ширину позволяет найти кратчайший путь из одной вершины графа до всех остальных вершин. Обход вершин ведется в очередности увеличения их удаления от корневого уровня.

Алгоритм А\* использует эвристику для быстрого нахождения кратчайшего пути между двумя вершинами в графе.

Оба алгоритма находят применение в широком спектре задач, но при выборе конкретного алгоритма необходимо учитывать особенности задачи и графа, на котором об будет использоваться.

Результатом данной работы является программа, которая способна находить самый оптимальный путь, используя алгоритмы А\* и поиск по ширине.

# **Список литературы**

1. A\* search algorithm[Электронный ресурс]: [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)(дата обращения: 29.04.2023).

2. Реализация алгоритма А\*[Электронный ресурс]:

<https://lsreg.ru/realizaciya-algoritma-poiska-a-na-c/> (дата обращения: 29.04.2023)

3. Алгоритм поиска по ширине[Электронный ресурс]:

<https://medium.com/nuances-of-programming/алгоритм-поиска-a-3bb59be05a79> (дата обращения: 29.04.2023)

4. Реализация алгоритмов поиска в ширину [Электронный ресурс]:

<https://ru.wikibooks.org/wiki/Реализации_алгоритмов/Поиск_в_ширину> (дата обращения: 29.04.2023)

5. Поиск в ширину-Алгоритмика[Электронный ресурс]:

<https://ru.algorithmica.org/cs/shortest-paths/bfs/> (дата обращения: 29.04.2023)

6. Руководство инсайдера по алгоритму A\* в Python. [Электронный ресурс]:<https://pythobyte.com/a-star-algorithm-python-93427/>.(дата обращения: 29.04.2023)

7. Реализация алгоритма A\*. [Электронный ресурс]: <https://habr.com/ru/articles/331220//>. ( дата обращения: 29.04.2023)

# **Приложения**

Приложение 1

Листинг программы

import math  
  
  
class Maze:  
 def \_\_init\_\_(self, filename, start, key, finish):  
 self.maze = []  
 self.start = start  
 self.key = key  
 self.finish = finish  
 self.get\_maze(filename)  
 self.width = len(self.maze[0])  
 self.height = len(self.maze)  
 self.add\_stuff()  
  
 def print\_maze(self):  
 for row in self.maze:  
 for element in row:  
 if element == 0:  
 print('#', end='')  
 elif element == 1:  
 print(' ', end='')  
 elif element == 2:  
 print('A', end='')  
 elif element == 3:  
 print('\*', end='')  
 elif element == 4:  
 print('E', end='')  
 elif element == 5:  
 print('.', end='')  
 elif element == 6:  
 print(',', end='')  
 elif element == 7:  
 print(';', end='')  
 print()  
 print()  
  
 def get\_maze(self, filename):  
 f = open(filename)  
 data = f.readlines()  
 self.maze = []  
 for i, row in enumerate(data):  
 self.maze.append([])  
 for element in row:  
 if element == '#':  
 self.maze[i].append(0)  
 else:  
 self.maze[i].append(1)  
  
 def check(self, dot):  
 y, x = dot[0], dot[1]  
 if y < 0 or x < 0 or y >= self.height or x >= self.width:  
 return False  
 return self.maze[y][x] > 0  
  
 def set(self, dot, number):  
 y, x = dot[0], dot[1]  
 if not self.check(dot):  
 return False  
 if self.maze[y][x] == 5:  
 self.maze[y][x] = 7  
 else:  
 self.maze[y][x] = number  
 return True  
  
 def add\_stuff(self):  
 flag = True  
 if not self.set(self.start, 2):  
 print("Игрок не выставлен - ячейка занята")  
 flag = False  
 if not self.set(self.key, 3):  
 print("Ключ не выставлен - ячейка занята")  
 flag = False  
 if not self.set(self.finish, 4):  
 print("Выход не выставлен - ячейка занята")  
 flag = False  
 if not flag:  
 exit(0)  
  
 def move(self, dot: tuple, direction=""):  
 y, x = dot[0], dot[1]  
 if direction == "up":  
 y -= 1  
 elif direction == "down":  
 y += 1  
 elif direction == "left":  
 x -= 1  
 elif direction == "right":  
 x += 1  
 return y, x  
  
 def create\_path(self, start, finish, prev):  
 current = finish  
 path = []  
 while current != start:  
 current = prev[current]  
 path.append(current)  
 path.pop()  
 path = path[::-1]  
 return path  
  
 def paint\_path(self, path, number):  
 for cell in path:  
 self.set(cell, number)  
  
 def width\_search(self, start, finish, number):  
 prev = {}  
 viewed = [start]  
 queue = [start]  
 directions = ["up", "down", "left", "right"]  
 while queue:  
 current = queue.pop(0)  
 for direction in directions:  
 y, x = self.move(current, direction)  
 if self.check((y, x)) and (y, x) not in viewed:  
 queue.append((y, x))  
 viewed.append((y, x))  
 prev[(y, x)] = current  
 if (y, x) == finish:  
 path = self.create\_path(start, finish, prev)  
 self.paint\_path(path, number)  
 return 0  
 print("Не найден путь")  
 exit(0)  
  
 def choose\_element(self, queue, finish):  
 y\_finish, x\_finish = finish[0], finish[1]  
 min\_value = 1e10  
 min\_element = queue[0]  
 for element in queue:  
 cell, dist = element[0], element[1]  
 y, x = cell[0], cell[1]  
 value = math.sqrt((y - y\_finish) \*\* 2 + (x - x\_finish) \*\* 2) + dist  
 if value < min\_value:  
 min\_element = element  
 queue.remove(min\_element)  
 return queue, min\_element  
  
 def a\_star\_search(self, start, finish, number):  
 prev = {}  
 viewed = [start]  
 queue = [(start, 0)]  
 directions = ["up", "down", "left", "right"]  
 while queue:  
 queue, element = self.choose\_element(queue, finish)  
 current, dist = element[0], element[1]  
 for direction in directions:  
 y, x = self.move(current, direction)  
 if self.check((y, x)) and (y, x) not in viewed:  
 queue.append(((y, x), dist + 1))  
 viewed.append((y, x))  
 prev[(y, x)] = current  
 if (y, x) == finish:  
 path = self.create\_path(start, finish, prev)  
 self.paint\_path(path, number)  
 return 0  
 print("Не найден путь")  
 exit(0)  
  
 def processing(self, method):  
 self.print\_maze()  
 if method == "width":  
 self.width\_search(self.start, self.key, 5)  
 self.width\_search(self.key, self.finish, 6)  
 else:  
 self.a\_star\_search(self.start, self.key, 5)  
 self.a\_star\_search(self.key, self.finish, 6)  
 self.print\_maze()  
  
  
def main():  
 maze = Maze("maze-for-u.txt", (0, 1), (107, 85), (199, 198))  
 maze.processing("width")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()