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 Старков А. В.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

**Содержание**

[Введение 3](#_Toc133789074)

[Цель работы 3](#_Toc133789075)

[Задачи 3](#_Toc133789076)

[Основная часть 4](#_Toc133789077)

[Алгоритмы поиска пути 4](#_Toc133789078)

[Алгоритм поиска в ширину 4](#_Toc133789079)

[Алгоритм А\* 5](#_Toc133789080)

[Структурное программирование 6](#_Toc133789081)

[Реализация алгоритма 7](#_Toc133789082)

[Пример работы алгоритма 11](#_Toc133789083)

[Заключение 13](#_Toc133789084)

[Список литературы 14](#_Toc133789085)

[Приложения 15](#_Toc133789086)

# Введение

В данной курсовой работе будет рассмотрено две важных темы в программировании: алгоритмы поиска пути и структурное программирование. Алгоритмы поиска пути являются фундаментальными в любой области, где необходимо находить оптимальные решения, такие как логистика, игры и т.д. Структурное программирование является подходом к написанию программ, который целью имеет улучшение читаемости, сопровождаемости и тестируемости кода. Будут изучены основные алгоритмы поиска пути, такие как алгоритм поиска в ширину и алгоритм A\*, а также принципы структурного программирования, такие как разделение программы на модули и использование функций. В результате работы будет разработана программа, использующая указанные алгоритмы поиска пути и реализующая принципы структурного программирования на языке Python.

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

## Задачи:

1. Изучить теорию алгоритмов обхода графа: в ширину и А\*;
2. Изучить, что такое структурное программирование;
3. Написать программу на языке Python для поиска маршрута в лабиринте с помощью алгоритма обхода в ширину и А\*.
4. Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.
5. Разобрать работу написанного кода, объяснить принцип работы алгоритмов и структуру программы.

# Основная часть

## Алгоритмы поиска пути

### Алгоритм поиска в ширину

Поиск в ширину (англ. breadth-first search) – это алгоритм поиска пути на графе, который работает в ширину, то есть просматривает все вершины на одном уровне, перед тем как продвинуться на следующий уровень. Алгоритм BFS начинает с начальной вершины, отмечает ее как посещенную и ставит ее в очередь. Затем он перебирает все соседние вершины, отмечает их как посещенные и также ставит их в очередь. Далее алгоритм извлекает следующую вершину из очереди и повторяет процесс для ее соседей. Алгоритм продолжает работу до тех пор, пока не будет достигнута конечная вершина или пока не будут просмотрены все вершины [1, с. 42-45].

Таким образом, алгоритм BFS обходит граф по слоям, начиная с начальной вершины. С каждой новой итерацией алгоритма он продвигается на один слой глубже в граф, пока не найдет нужную вершину или не проверит все вершины на всех уровнях [7, с. 640].

Для того чтобы алгоритм BFS работал правильно, необходимо использовать очередь для хранения вершин, которые еще не были посещены. Кроме того, необходимо отметить каждую вершину как посещенную во время ее добавления в очередь, а не при ее извлечении из очереди. Это гарантирует, что каждая вершина будет рассмотрена только один раз.

Алгоритм BFS автоматически находит кратчайший путь между начальной и конечной вершинами, так как он обходит вершины в порядке их удаления от начальной вершины. Если две вершины находятся на одном уровне (т.е. имеют одинаковое расстояние от начальной вершины), то алгоритм выберет ту вершину, которая была добавлена в очередь раньше.

Алгоритм BFS применяется для поиска кратчайшего пути в невзвешенном графе, а также для поиска всех доступных вершин на определенном расстоянии от начальной вершины. Он также может использоваться для проверки наличия циклов в графе и для нахождения связных компонент графа [3].

### Алгоритм А\*

Алгоритм A\* известен как эффективный метод поиска субоптимального маршрута от начальной до конечной вершины с наименьшей стоимостью. Он использует эвристическую функцию "расстояние + стоимость" (f(x) = g(x) + h(x)), где функция h(x) представляет собой допустимую эвристическую оценку, которая не переоценивает расстояние к целевой вершине. Обычно она определяется как расстояние по прямой до цели, поскольку это является наименьшим расстоянием между двумя точками.

Алгоритм A\* является информированным алгоритмом поиска маршрутов от начальной вершины до конечной. 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 выборе вершины, A\* учитывает не только стоимость текущего маршрута, но также и весь пройденный путь от начальной вершины (g(x) — это стоимость пути от начальной позиции, а не от предыдущей, как в жадном алгоритме). A\* продолжает поиск, пока не найдет наименьший маршрут от начальной до конечной вершины [6].

Алгоритм A\* является полным, т. е. всегда находит решение, если оно существует. Он также оптимально эффективен для заданной эвристики h. Это значит, что любой другой алгоритм исследует не меньше узлов, чем алгоритм A\* (за исключением случаев, когда существует несколько частных решений с одинаковой эвристикой, точно соответствующей стоимости оптимального пути).

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 [2].

## 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 [4].

Структурное программирование стало популярным в 1970-х годах, когда были разработаны языки программирования, такие как ALGOL, FORTRAN, COBOL. Эти языки предлагали программистам использовать только ограниченное количество структур управления и отказаться от операторов типа GOTO, которые могли бы создать запутанный и трудночитаемый код [8].

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.

Еще один плюс структурного программирования заключается в возможности проектировать программы, используя метод декомпозиции. Крупные программы могут быть разбиты на небольшие логические блоки, каждый из которых отвечает за свою определенную функцию. Программисты могут работать над каждым блоком независимо и тестируя его отдельно от других, затем объединить их в работающую программу.

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

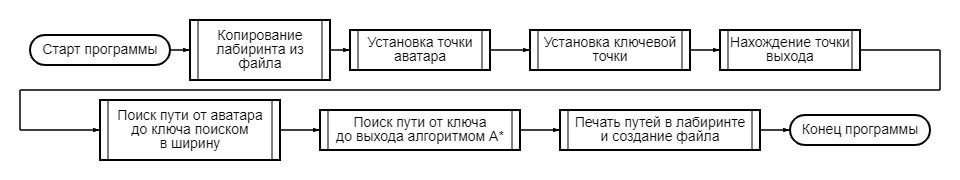
Реализацию алгоритма стоит начать с разработки структуры. В данной программе должна быть функция, при запуске которой будет работать связно вся программа. В этой функции будет происходить копирование лабиринта из файла, установка точки аватара (начальной точки), ключевой точки, нахождения точки выхода, последовательный вызов алгоритмов поиска пути, перенос траектории пути в лабиринт и последующее создание файла с этими траекториями. Под каждое действие будет создана своя функция для меньшего загромождения главной функции и модульности программы, что облегчит разработку и отладку. Для наглядности сделаем схему работы алгоритма (Рис. 1).

Рисунок 1. Схема работы алгоритма

Теперь приступим непосредственно к написанию кода программы (Приложение 1).

Напишем функцию main(), которая будет использовать такие функции как:

- read\_maze(fileMaze="maze-for-u.txt") - функция чтения файла с лабиринтом;

- sel\_point(maze, code) - функция для выбора местоположения ключа или аватара;

- sel\_exit(maze) - функция для выбора местоположения выхода на лабиринте;

- bfs(start, end, maze) - функция поиска в ширину от начальной точки до ключа в лабиринте;

- astar(maze, start, end) - функция поиска алгоритмом A\* от ключа до выхода в лабиринте;

- createFile(maze, path\_bfs, path\_astar, key) - функция создания файла с итоговыми результатами работы программы.

В программе также есть вспомогательные функции:

- near(x, y, maze) - функция отображения окрестности точки;

- get\_moves(pos,maze) – функция для определения возможных путей из текущей позиции;

- heuristic(a, b) – внутренняя функция функции astar() для нахождения эвристического расстояния;

- get\_key(val,my\_dict) - внутренняя функция функции astar() для возвращения ключа словаря по значению.

Разберём каждую из этих функций.

Функция read\_maze() считывает файл с лабиринтом и переводит его в удобный для программы формат.

Кодовые замены, выполняемые здесь, заменяют символы в лабиринте:

- "#" на "1", что означает наличие стены в указанной точке.

- " " на "0", что означает отсутствие стены в указанной точке.

Функция sel\_point() запрашивает у пользователя координаты точки аватара, если аргумент code равен 1, или ключа, если аргумент code равен 2.

Вводятся последовательно координаты X и Y. После программа проверяет значение в матрице лабиринта по данным координатам. Если значение равно «1», то на данном месте стена и программа сообщает об этом и выводит окружение данной точки с помощью вспомогательной функции near(). Если значение в матрице равно 0, то программа возвращает координаты аватара в виде кортежа (x, y).

В случае некорректного ввода значений, программа выводит соответствующее предупреждение и вновь запрашивает ввод координат.

Функция sel\_exit() находит на нижней границе лабиринта точку не являющейся стеной. Так как она единственная, то она и будет являться выходом.

Функция bfs() реализует поиск в ширину (BFS) в лабиринте.

Для упрощения работы этой функции импортируем класс Queue из модуля queue.

Класс Queue - это реализация очереди (FIFO - первым пришел, первым обслужен) в Python.

После импорта мы можем создать объект очереди, добавлять элементы в очередь и получать элементы из очереди.

Переменная queue - очередь для выполнения BFS, начинается с начальной точки и пути до неё.

Основная логика заключается в цикле while, который выполняется до тех пор, пока очередь не пуста. На каждой итерации извлекается первый элемент из очереди, который представляет собой текущую позицию в лабиринте и путь до неё. Затем проверяется, является ли эта позиция конечной. Если да, то возвращается найденный путь до неё.

Если текущая позиция не является конечной, то выполняется цикл for, который перебирает все возможные ходы (направления, доступные из текущей позиции, с помощью функции get\_moves(), которая проверяет, можно ли двигаться вверх, вниз, влево или вправо от текущей позиции и добавляет новую позицию в список ходов, если это возможно). Для каждого хода проверяется, что он не находится в пути, чтобы избежать циклических путей. Если это так, то позиция добавляется в очередь вместе с обновленным путём до неё (добавляя новую позицию в конец пути).

Если цикл for завершится, а конечная позиция не была достигнута, то функция вернёт None.

Функция astar() реализует поиск пути в лабиринте с помощью алгоритма A\*.

Сначала определяются размеры лабиринта (количество строк и столбцов), а также возможные направления движения (вправо, вниз, влево, вверх).

Затем определяется эвристическая функция heuristic(), которая используется в дальнейшем для оценки расстояния от каждой точки до конечной точки.

Создаются два списка для открытых и закрытых вершин, а также словарь, где для каждой точки будет храниться значение f = g + h, где g - стоимость пути от начальной точки до данной, а h - эвристическое расстояние от данной точки до конечной.

Далее происходит итерация, пока есть открытые вершины. На каждой итерации находится вершина с минимальным значением f. Если найдена конечная точка, создается путь от нее до начальной точки с помощью цикла, который ищет предыдущие точки, чтобы восстановить путь.

Если конечная точка не найдена, то возвращается значение None.

Функция createFile() создает файл с лабиринтом, в который заносятся пути от аватара до ключа и от ключа до выхода.

В первом блоке заменяются все точки лабиринта, соответствующие пути алгоритма BFS, на символ ".". Затем, во втором блоке заменяются все точки лабиринта, соответствующие пути алгоритма А\*, на символ ",". В третьем блоке данный алгоритм заменяет точку, где находился ключ, на символ "\*".

Наконец, данный алгоритм создает файл "maze-for-me-done.txt" и записывает измененный лабиринт туда. Если все прошло успешно, функция выводит сообщение о том, что лабиринт успешно пройден и получившийся путь находится в файле "maze-for-me-done.txt". Если произошла какая-то ошибка при поиске пути, функция выводит сообщение об ошибке.

Программа запускается вызовом функции main().

# Пример работы алгоритма

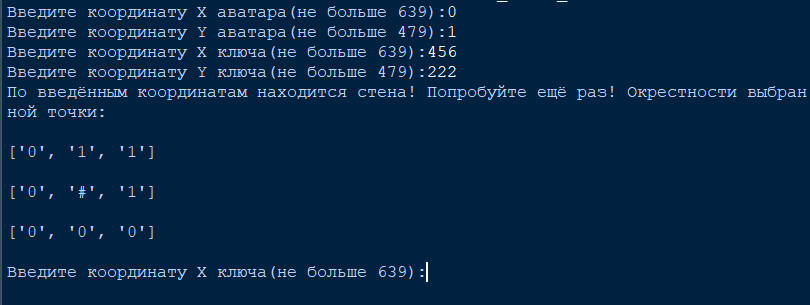
При запуске программы она просит ввести координаты аватара и ключа. Если введённые координаты соответствуют стене, то выводятся окрестности этой точки для упрощения повторного выбора точки (Рис. 2).

Рисунок 2. Обработка введённых координат, соответствующих стене

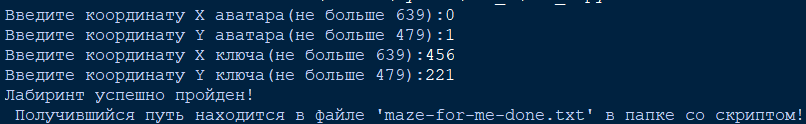
Если все координаты введены корректно, то программа находит путь и оповещает об этом (Рис. 3).

Рисунок 3. Успешная работа программы

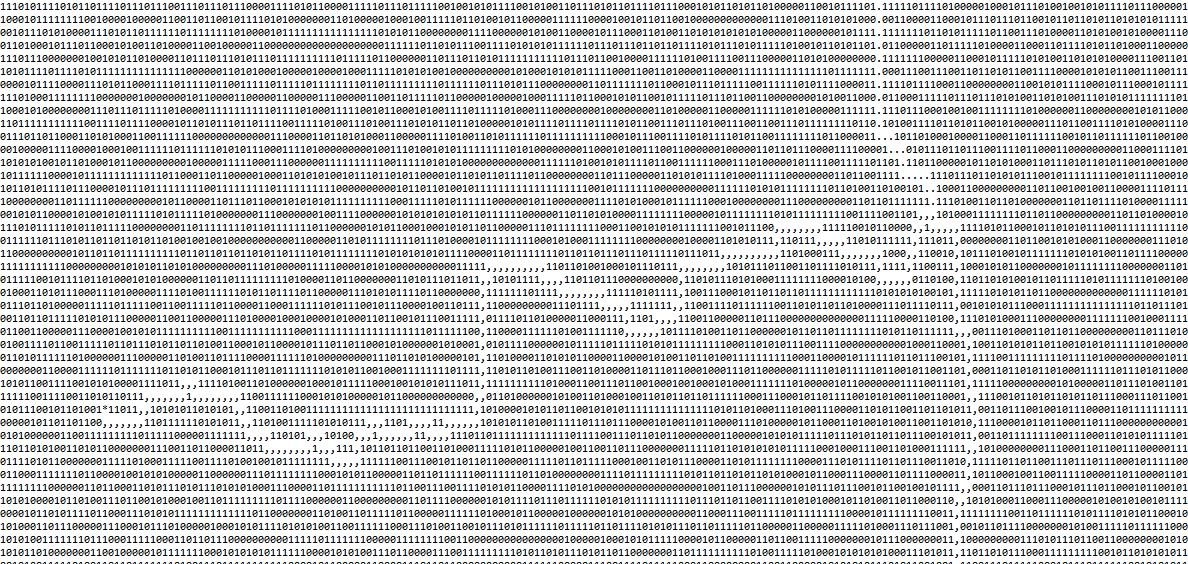
Файл «maze-for-me-done.txt», получившийся в результате работы алгоритма, представляет собой лабиринт, где «1» - стена, «0» - коридор, «.» - траектория пути алгоритма BFS, «,» - траектория пути алгоритма А\*, «\*» - ключ. Так как содержимое файла имеет 640 строк и 480 столбцов, в качестве примера работы будет продемонстрирован участок у ключевой точки при введённых координатах аватара (0,1) и ключа (456, 221) (Рис. 4).

Рисунок 4. Участок файла

# Заключение

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

Написана программа на языке Python, реализующая алгоритмы обхода графа в ширину и А\* для поиска маршрута в лабиринте. Результаты обходов лабиринта и получившиеся маршруты были сохранены в файл.

Структура программы состоит из модулей для работы с лабиринтом, а также модулей для реализации алгоритмов обхода в ширину и А\*. Число строк кода программы составляет примерно 200 строк.

Таким образом, цель курсовой работы достигнута. Результатом является программа, способная находить маршрут в лабиринте с помощью алгоритмов обхода графа в ширину и А\*, с сохранением полученных маршрутов в файле. Работа программы была проверена на нескольких тестовых лабиринтах и показала хорошие результаты.

# Список литературы

1. Алексеев В.Е., Таланов В.А. «Графы. Модели вычислений. Структуры данных» - Нижний Новгород: Издательство Нижегородского госуниверситета, 2004. – 291 с.
2. Басараб М.А., Домрачева А.Б., Купляков В.М. Алгоритмы решения задачи быстрого поиска пути на географических картах. / Басараб М.А., Домрачева А.Б., Купляков В.М. - Текст: электронный // Инженерный журнал: наука и инновации. - 2013. - № 11. – с. 8 - URL: <https://cyberleninka.ru/article/n/algoritmy-resheniya-zadachi-bystrogo-poiska-puti-na-geograficheskih-kartah/viewer>
3. Гришутин А., Алексеев С., Иванов М., «Алгоритмика - Поиск в ширину» [Электронный ресурс]. URL: <https://ru.algorithmica.org/cs/shortest-paths/bfs/> (Дата обращения: 29.04.2023)
4. Лекция 10. Структурное программирование, предпрограммная подготовка задачи// StudFiles / [Электронный ресурс]. URL: <https://studfile.net/preview/4290050/> (Дата обращения: 29.04.2023)
5. Структурное программирование сверху-вниз и правильность программ // Allbest/ [Электронный ресурс]. URL: <https://revolution.allbest.ru/programming/00712728_0.html> (Дата обращения: 29.04.2023)
6. Easy A \* (звезда) Поиск пути // digitrain.ru / [Электронный ресурс]. URL: <https://digitrain.ru/articles/337034/> (Дата обращения: 29.04.2023)
7. Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser. «Structures and Algorithms in Java. Sixth Edition» - United States of America: John Wiley & Sons, Inc., 2014 г. – 720 с.
8. What is Structured Programming? // techopedia / [Электронный ресурс]. URL: <https://www.techopedia.com/definition/16413/structured-programming> (Дата обращения: 29.04.2023)

# Приложения

**Приложение 1**

**Листинг программы**

from queue import Queue

# Функция для импорта и преобразования исходного файла

def read\_maze(fileMaze="maze-for-u.txt"):

maze = [list(line.replace("#", "1").replace(" ", "0")) for line in open(fileMaze).read().split("\n")[:-1]]

return maze

# Функция отображения окрестности точки

def near(x, y, maze):

near = []

if y == 0:

if x == 0:

near = [["#", maze[x][y + 1]],

[maze[x + 1][y], maze[x + 1][y + 1]]]

elif x == (len(maze) - 1):

near = [[maze[x - 1][y], maze[x - 1][y + 1]],

["#", maze[x][y + 1]]]

else:

near = [[maze[x - 1][y], maze[x - 1][y + 1]],

["#", maze[x][y + 1]],

[maze[x + 1][y], maze[x + 1][y + 1]]]

elif y == (len(maze[0]) - 1):

if x == 0:

near = [[maze[x][y - 1], "#"],

[maze[x + 1][y - 1], maze[x + 1][y]]]

elif x == (len(maze) - 1):

near = [[maze[x - 1][y - 1], maze[x - 1][y]],

[maze[x][y - 1], "#"]]

else:

near = [[maze[x - 1][y - 1], maze[x - 1][y]],

[maze[x][y - 1], "#"],

[maze[x + 1][y - 1], maze[x + 1][y]]]

elif x == 0:

near = [[maze[x][y - 1], "#", maze[x][y + 1]],

[maze[x + 1][y - 1], maze[x + 1][y], maze[x + 1][y + 1]]]

elif x == (len(maze) - 1):

near = [[maze[x - 1][y - 1], maze[x - 1][y], maze[x - 1][y + 1]],

[maze[x][y - 1], "#", maze[x][y + 1]]]

else:

near = [[maze[x - 1][y - 1], maze[x - 1][y], maze[x - 1][y + 1]],

[maze[x][y - 1], "#", maze[x][y + 1]],

[maze[x + 1][y - 1], maze[x + 1][y], maze[x + 1][y + 1]]]

return near

# Функция определения точки аватара (начальной точки) или ключа

def sel\_point(maze, code):

if code == 1:

word = "аватара"

if code == 2:

word = "ключа"

while 1:

try:

x = int(input(f"Введите координату X {word}(не больше {len(maze) - 1}):"))

if x >= 0 and x < len(maze):

y = int(input(f"Введите координату Y {word}(не больше {len(maze[0]) - 1}):"))

if y >= 0 and y < len(maze[0]):

if str(maze[x][y]) == "1":

print(

"По введённым координатам находится стена! Попробуйте ещё раз! Окрестности выбранной точки:\n")

for line in near(x, y, maze):

print(line, "\n")

else:

return (x, y)

else:

print("Введённое значение выходит за допустимый диапазон! Попробуйте ещё раз.")

continue

else:

print("Введённое значение выходит за допустимый диапазон! Попробуйте ещё раз.")

continue

except Exception:

print("Вы ввели неверное значение! Попробуйте ещё раз.")

# Функция нахождения точки выхода

def sel\_exit(maze):

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

if int(maze[len(maze)-1][y]) == 0:

return (len(maze) - 1,y)

# Функция для определения возможных путей из текущей позиции

def get\_moves(pos,maze):

moves = []

if pos[0] > 0 and not int(maze[pos[0] - 1][pos[1]]):

moves.append((pos[0] - 1, pos[1]))

if pos[0] < len(maze) - 1 and not int(maze[pos[0] + 1][pos[1]]):

moves.append((pos[0] + 1, pos[1]))

if pos[1] > 0 and not int(maze[pos[0]][pos[1] - 1]):

moves.append((pos[0], pos[1] - 1))

if pos[1] < len(maze[0]) - 1 and not int(maze[pos[0]][pos[1] + 1]):

moves.append((pos[0], pos[1] + 1))

return moves

# Функция для поиска пути с помощью поиска в ширину

def bfs(start, end, maze):

queue = Queue()

queue.put((start, [start]))

while not queue.empty():

pos, path = queue.get()

if pos == end:

return path

for move in get\_moves(pos,maze):

if move not in path:

queue.put((move, path + [move]))

return None

# Функция поиска пути в лабиринте с помощью алгоритма A\*

def astar(maze, start, end):

# Определяем размеры лабиринта

rows = len(maze)

cols = len(maze[0])

# Определяем набор возможных направлений движения (вправо, вниз, влево, вверх)

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

# Определяем эвристическую функцию

def heuristic(a, b):

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

# Создаем два списка для открытых и закрытых вершин

# Каждая вершина представляется в виде кортежа (y,x)

open = [start]

close = []

# Создаем словарь, где для каждой точки будет храниться значение f = g + h, где g - стоимость пути от начальной точки до данной,

# а h - эвристическое расстояние от данной точки до конечной

g = {start: 0}

f = {start: heuristic(start, end)}

# Пока есть открытые вершины

while open:

# Находим вершину с минимальным значением F

current = min(open, key=lambda x: f[x])

# Если найдена конечная точка

if current == end:

# Создаем путь

def get\_key(val,my\_dict):

for key, value in my\_dict.items():

if val == value:

return key

path = []

i=g[current]

path.append(current)

while i > 0:

maybe = get\_key(i - 1,g)

if ((current[0]-maybe[0])\*\*2 + (current[1]-maybe[1])\*\*2)\*\*(1/2) == 1:

current = maybe

path.append(maybe)

i-=1

else:

del g[maybe]

path.reverse()

return path

# Переносим текущую вершину из открытых в закрытые

open.remove(current)

close.append(current)

# Проходимся по всем возможным направлениям движения

for dir in dirs:

# Вычисляем новую точку

new\_point = (current[0] + dir[0], current[1] + dir[1])

# Проверяем, что новая точка находится в пределах лабиринта и не является стеной

if new\_point[0] >= 0 and new\_point[0] < rows and new\_point[1] >= 0 and new\_point[1] < cols and int(maze[new\_point[0]][new\_point[1]]) == 0:

# Вычисляем стоимость пути от начальной точки до новой точки

temp\_g = g[current] + 1

# Если новая точка уже была рассмотрена, проверяем, что текущий путь до нее лучше, чем предыдущий

if new\_point in g:

if temp\_g < g[new\_point]:

g[new\_point] = temp\_g

else:

continue

# Если новая точка еще не была рассмотрена, добавляем ее в открытые вершины

else:

g[new\_point] = temp\_g

h = heuristic(new\_point, end)

f[new\_point] = g[new\_point] + h

open.append(new\_point)

# Если путь не найден

return None

def createFile(maze,path\_bfs,path\_astar,key):

# Заносим изменения в лабиринт

try:

# Заносим путь от аватара до ключа

for point in path\_bfs:

maze[point[0]][point[1]] = "."

# Заносим путь от ключа до выхода

for point in path\_astar:

maze[point[0]][point[1]] = ","

# Заносим ключевую точку

maze[key[0]][key[1]] = "\*"

# Генерируем файл с пройдённым лабиринтом

with open("maze-for-me-done.txt", "w") as f:

f.write('\n'.join([''.join(map(str, line)) for line in maze]))

print("Лабиринт успешно пройден!\n Получившийся путь находится в файле 'maze-for-me-done.txt' в папке со скриптом!")

except Exception:

print("При поиске пути произошла ошибка!")

def main():

# Создание лабиринта, нахождение габаритов

try:

maze = read\_maze()

except Exception:

print("Исходный файл не найден!")

# Объявление и инициализация точек

start = sel\_point(maze, 1)

key = sel\_point(maze, 2)

end = sel\_exit(maze)

# Запускаем поиск в ширину от аватара до ключа

path\_bfs = bfs(start, key, maze)

# Запускаем поиск алгоритмом A\* от ключа до выхода

path\_astar = astar(maze, key, end)

createFile(maze,path\_bfs,path\_astar,key)

main()