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 Кузнецов Н.И.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc136289558)

[Задачи 3](#_Toc136289559)

[1.Теоретическая часть 5](#_Toc136289560)

[а) Поиск в ширину 5](#_Toc136289561)

[б) Алгоритм А\* 7](#_Toc136289562)

[в) Структурное программирование 10](#_Toc136289563)

[2. Реализация алгоритма 12](#_Toc136289564)

[Пример работы 15](#_Toc136289565)

[Заключение 17](#_Toc136289566)

[Список литературы 18](#_Toc136289567)

[Приложение 1 19](#_Toc136289568)

[Листинг программы 19](#_Toc136289569)

# Введение

Алгоритмы поиска пути являются важной частью компьютерных наук и находят применение во многих областях, включая компьютерные игры, маршрутизацию в компьютерных сетях и робототехнику. Структурное программирование — это технология создания программ, которая предполагает использование только трех основных структур управления: последовательности, ветвления и цикла, позволяющая путем соблюдения определенных правил уменьшить время разработки и количество ошибок, а также облегчить возможность модификации программы. Структурный подход охватывает все стадии разработки проекта: спецификацию, проектирование, собственно программирование и тестирование [1]. Одними из этих алгоритмов являются поиск в ширину и А\*.

Целью данной курсовой работы является изучение и реализация алгоритмов поиска пути и их взаимосвязи со структурным программированием. В рамках работы будут рассмотрены два ключевых алгоритма: поиск в ширину (BFS - Breadth-First Search) и алгоритм A\* (A-star). Оба алгоритма являются эффективными инструментами для нахождения оптимальных путей в графах с различными характеристиками.

**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, 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

## а) Поиск в ширину

Поиск в ширину (BFS - Breadth-First Search) является одним из фундаментальных алгоритмов обхода графа, который широко применяется в программировании и анализе данных. Он используется для нахождения кратчайшего пути от исходной вершины до всех достижимых вершин в невзвешенном графе или для поиска определенного элемента в графе.

Принцип работы алгоритма BFS основан на обходе графа "в ширину", то есть на просмотре вершин на одном уровне перед переходом к следующему уровню. Алгоритм начинает с исходной вершины и постепенно расширяет свою область поиска, путешествуя по смежным вершинам.

Один из ключевых компонентов алгоритма BFS - использование очереди для хранения вершин, которые нужно посетить. При обработке вершины она извлекается из очереди, а ее соседние не посещённые вершины добавляются в конец очереди. Это гарантирует, что вершины будут посещены в порядке их удаленности от исходной вершины, что соответствует обходу графа "в ширину".

Алгоритм BFS также требует отслеживания посещенных вершин, чтобы избежать зацикливания и повторного посещения вершин. При посещении вершины она помечается как посещенная, что позволяет избежать повторного добавления в очередь.

Основной результат выполнения алгоритма BFS - получение дерева обхода, которое представляет собой дерево связей между вершинами. Данное дерево позволяет наглядно представить кратчайшие пути от исходной вершины до всех достижимых вершин в графе.

Временная сложность алгоритма BFS составляет O (V + E), где V - количество вершин, а E - количество ребер в графе. Это обеспечивает достаточно эффективную работу алгоритма даже для больших графов.

Алгоритм BFS находит применение во множестве задач, включая нахождение кратчайшего пути в невзвешенном графе, проверку связности графа, поиск компонент связности, поиск циклов и другие [2].

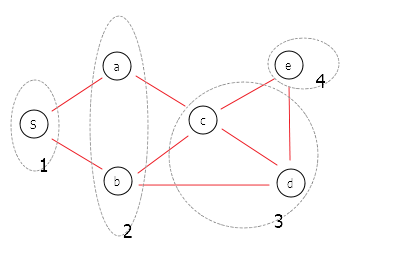


Рисунок 1 – Поиск в ширину на графе

## б) Алгоритм А\*

Алгоритм A\* (A-star) является эффективным алгоритмом поиска пути, который широко применяется для нахождения оптимального пути от исходной вершины до целевой вершины на географических картах и других графах с весами. Он сочетает в себе принципы поиска в ширину (BFS) и эвристическую оценку для определения наиболее перспективных путей.

Алгоритм A\* использует эвристическую функцию, которая оценивает оставшуюся стоимость до цели из текущей вершины. Эта функция обычно обозначается как h(n). Важно отметить, что эвристическая функция должна быть допустимой, то есть не переоценивать стоимость пути до целевой вершины. Если эвристическая функция является допустимой, то алгоритм A\* гарантирует нахождение оптимального пути.

Процесс работы алгоритма A\* может быть представлен следующим образом:

1) Создать список открытых вершин и поместить в него исходную вершину.

2) Создать список закрытых вершин.

3) Пока список открытых вершин не опустеет или не будет достигнута целевая вершина, выполнить следующие шаги:

а) Выбрать вершину с наименьшим значением f(n) = g(n) + h(n) из списка открытых вершин, где g(n) - стоимость пути от исходной вершины до текущей вершины.

б) Поместить выбранную вершину в список закрытых вершин.

в) Раскрыть соседей выбранной вершины и вычислить для них значения g(n), h(n) и f(n).

г) Добавить раскрытых соседей в список открытых вершин, если они еще не присутствуют в списке или обновить их значения g(n), h(n) и f(n), если они уже есть в списке.

4) Если целевая вершина была достигнута, построить путь от исходной вершины до целевой вершины, используя информацию о родительских вершинах.

Алгоритм A\* обладает оптимальностью, так как использует эвристическую функцию для оценки оставшейся стоимости и эффективно выбирает наиболее перспективные пути для раскрытия. Он также является информированным алгоритмом, поскольку использует эвристическую информацию о графе [3].

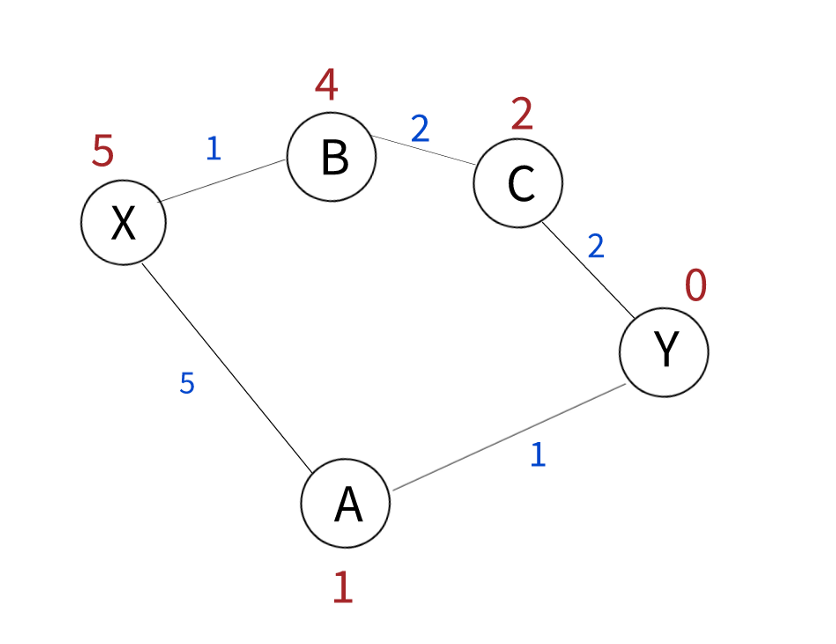


Рисунок 2 – Алгоритм А\* на графе

Допустим, мы хотим попасть из точки X в точку Y. Так как вершина Х не меняет своего положения, мы можем отбросить g(n) — ее значение равно 0. Эвристическое значение этой вершины выделено красным шрифтом — 5.

В подобных задачах эвристическое значение — стоимость достижения рассматриваемой вершины из начальной.

Из вершины Х есть два пути.

Если мы перейдем в вершину А, g(n) будет равна 5 (стоимость пути), так как мы перемещаемся в новую вершину. Значение h(n) теперь равно 1. Значение f(n) в точке А будет равно 5+1 = 6. Теперь найдем значение f(n) каждой точки:

X— A => g(A) + f(A) = 5 + 1 = 6,

A — Y=> g(Y) + f(Y) = 6+ 0= 6,

X— B => g(B) + f(B) = 1+ 4= 5,

B — C => g(C) + f(C) = 3+ 2= 5,

C — Y=> g(Y) + f(Y) = 5 + 0= 5,

Как видно из наших вычислений, кратчайший путь — X-B-C-Y. Его стоимость равна 5, в то время как X-A-Y — 6.

## в) Структурное программирование

Структурное программирование представляет собой методологию разработки программного кода, основанную на идее разделения программы на логические структуры, такие как последовательность, условие и цикл. Основной целью структурного программирования является упрощение понимания программного кода, его сопровождения и отладки.

Одним из ключевых принципов структурного программирования является использование только трех основных структур: последовательность (sequence), ветвление (selection) и цикл (iteration). Последовательность позволяет выполнять операции последовательно, ветвление позволяет выбирать между различными вариантами выполнения кода, а цикл позволяет повторять операции необходимое количество раз. Эти структуры могут быть комбинированы для создания сложных программных решений.

Структурное программирование подчеркивает важность использования блок-схем и псевдокода при проектировании программы. Блок-схемы позволяют наглядно представить логику выполнения программы и взаимосвязь между различными структурами. Псевдокод является упрощенным текстовым представлением программы, не зависящим от конкретного языка программирования, что упрощает понимание и коммуникацию между разработчиками.

Структурное программирование также подчеркивает важность использования модульного подхода при разработке программного кода. Программа разделяется на небольшие, независимые модули, каждый из которых выполняет конкретную функцию. Это позволяет повысить читабельность и переиспользуемость кода, а также упростить его тестирование и отладку.

Одним из преимуществ структурного программирования является устранение проблемы "спагетти-кода", которая возникает при использовании неструктурированных подходов. Структурированное программирование обеспечивает более легкое понимание программы и ее модификацию, а также улучшает управляемость и надежность программного кода [4].

# 2. Реализация алгоритма

Для начала создадим функцию read\_maze, которая принимает входной текстовый файл с лабиринтом и возвращает его в виде двумерного массива maze. Затем определим функцию neighbors, которая принимает лабиринт maze в качестве аргумента и возвращает список valid\_neighbors, содержащий кортежи с координатами доступных ячеек, в которые можно переместиться. Далее разработаем алгоритм поиска в ширину.

Создадим функцию find\_path. Входным параметром функции будет сам лабиринт maze. В начале укажем точки старта и конца поиска в переменных start и end соответственно.

Создадим очередь queue, в которую добавим кортеж (start, [start]). В этом кортеже первый элемент - координаты начальной точки, а второй - путь к этой точке.

Затем создадим множество visited, в которое будем добавлять посещенные точки. Запустим цикл, который будет выполняться, пока очередь queue не станет пустой. На каждой итерации цикла извлечем из очереди первый элемент current и путь path к нему. Если текущая точка current равна конечной точке end, это означает, что путь найден, и мы возвращаем его. В противном случае добавляем точку current в множество visited, а затем для каждого соседа neighbor текущей точки вызываем функцию neighbors.

Для каждого из этих соседей проверяем, не посещали ли мы его ранее. Если он не был посещен, добавляем в очередь queue кортеж (neighbor, path + [neighbor]), где neighbor - это координаты соседней точки, а path + [neighbor] - это путь к ней. Поиск завершается, когда весь лабиринт будет пройден. Если точка end не была найдена, функция возвращает None.

После этого разработаем алгоритм A\*. Создадим функцию heuristic, которая будет вычислять эвристическое расстояние от текущей ячейки до конечной точки в алгоритме A\*. Входными параметрами функции будут координаты текущей ячейки cell и координаты конечной точки end. Функция будет вычислять расстояние между текущей ячейкой и конечной точкой с помощью формулы Евклида: sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2), где cell[0] и cell[1] - координаты текущей ячейки, а end[0] и end[1] - координаты конечной точки. (Эвристическое расстояние используется для оценки стоимости пути от текущей ячейки до конечной точки. Оно добавляется к фактической стоимости пути, чтобы получить общую стоимость пути от начальной точки до конечной. В результате A\* выберет путь с наименьшей общей стоимостью.)

Создадим функцию find\_path\_a\_star. Входным параметром функции будет сам лабиринт maze. Сначала определим начальную и конечную точки start и end. Затем создадим приоритетную очередь queue, в которую добавим кортеж, содержащий стоимость пути, текущую точку и путь к текущей точке.

Затем создадим множество visited, в котором будут храниться посещенные точки, чтобы избежать повторных посещений. В цикле while, пока очередь не станет пустой, извлечем кортеж с минимальной стоимостью из приоритетной очереди. Затем проверим, является ли текущая точка конечной точкой. Если да, то функция вернет стоимость пути и путь к конечной точке. Если текущая точка не является конечной, добавим ее в множество visited, чтобы не посещать ее повторно. Затем для каждого соседа текущей точки вызовем функцию neighbors. Если соседняя точка не была посещена ранее, создадим новый путь к этой точке, добавив ее к пути к текущей точке. Затем вычислим стоимость нового пути, используя формулу priority = len(new\_path) + heuristic(neighbor, end). Здесь len(new\_path) - фактическая стоимость пути от начальной точки до текущей, а heuristic(neighbor, end) - эвристическое расстояние от соседней точки до конечной точки. Общая стоимость нового пути вычисляется как сумма фактической стоимости пути и эвристического расстояния.

Cоздадим новый кортеж с общей стоимостью, соседней точкой и новым путем, и добавим его в приоритетную очередь. Это продолжится до тех пор, пока конечная точка не будет достигнута или пока приоритетная очередь не опустеет. Если конечная точка не может быть достигнута из начальной точки, функция вернет None.

# Пример работы

Программа находит путь, и, если работает верно, сообщает об этом. Результат работы алгоритма сохраняется в файле "maze-for-me-done.txt", который представляет собой лабиринт с определенными символами: "." - траектория пути алгоритма поиска в ширину, "," - траектория пути алгоритма А. Для наглядности, будет продемонстрирован результат работы программы:

Алгоритм А\*:

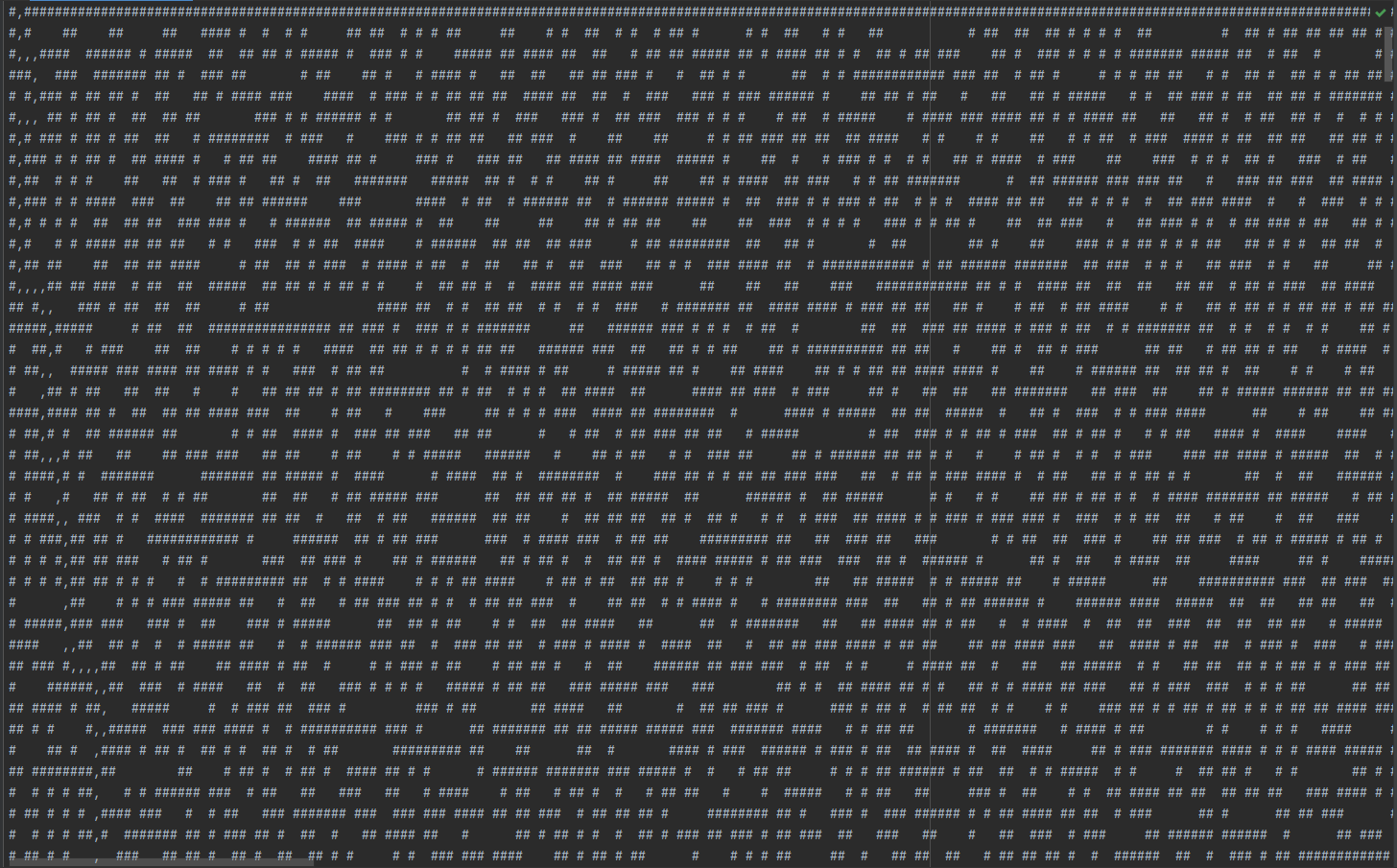


Рисунок 3 – Результат работы программы по алгоритму А\*

Поиск в ширину:

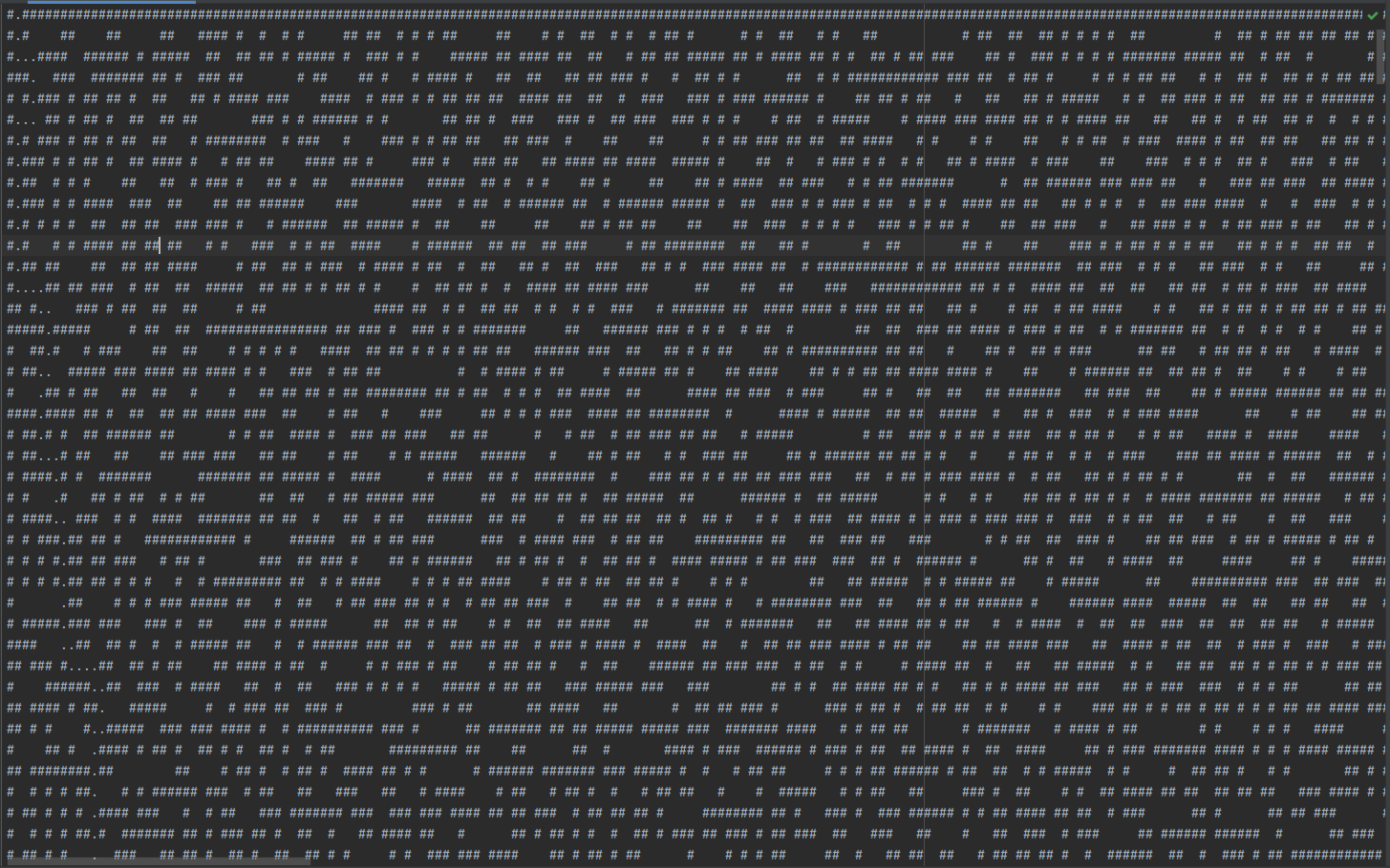


Рисунок 4 – Результат работы программы по алгоритму поиска в ширину

# Заключение

В ходе выполнения данной задачи были изучены теоретические основы алгоритмов прохода по графу в ширину и А\*, а также структурного программирования.

Основная разница между А\* и поиском в ширину заключается в их стратегии поиска. Поиск в ширину исследует все возможные пути равномерно, тогда как А\* использует информацию о стоимости пути и эвристическую оценку, чтобы более эффективно сосредоточиться на перспективных путях к цели.

Создана программа на языке Python, реализующая алгоритмы прохода по графу в ширину и А\* для поиска оптимального пути в лабиринте. Результаты проходов по лабиринту и полученные пути были сохранены в файле.

Результатом является программа, способная находить оптимальный путь в лабиринте с использованием алгоритмов прохода по графу в ширину и А\*, а также сохранять полученные пути в файле.

# Список литературы

1. Авачева Т. Г., Пруцков А. В. Современный взгляд на концепцию структурного программирования (рус.) // Cloud of Science : Журнал. — 2019. — Т. 6, № 4. — С. 646–665. (Дата обращения 26.05.2023)
2. Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Структуры данных: Учебник. – Нижний Новгород: Изд-во ННГУ, 2005. 307 с. (Дата обращения 28.05.2023)
3. 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 (Дата обращения 23.05.2023)
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 26.05.2023)
5. Лекция 10. Структурное программирование, предпрограммная подготовка задачи// StudFiles / [Электронный ресурс]. URL: <https://studfile.net/preview/4290050/> (Дата обращения: 14.05.2023)
6. Лингер Р., Миллс X., Уитт Б. Теория и практика структурного программирования: Пер. с англ. М.: Мир, 1982.— 406 с., ил. С. 324-327. (Дата обращения 26.05.2023)
7. Томас Кормен, и др. «Алгоритмы: построение и анализ», 3-е изд.: Пер. с англ. – М.: ООО “И. Д. Вильямс” 2013. – 630 с. (Дата обращения 14.05.2023)

# Приложение 1

## Листинг программы

from queue import PriorityQueue

from math import sqrt

def read\_maze(filename):

with open(filename) as f:

maze = [[char for char in line.strip()] for line in f]

return maze

def 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 ширину

def find\_path(maze):

start = (0, 1)

end = (len(maze) - 1, len(maze[0]) - 2)

queue = [(start, [start])]

visited = set()

while queue:

current, path = queue.pop(0)

if current == end:

return path

visited.add(current)

for neighbor in neighbors(maze, current):

if neighbor not in visited:

queue.append((neighbor, path + [neighbor]))

return None

# А\*

def heuristic(cell, end):

return sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2)

def find\_path\_a\_star(maze):

start = (0, 1)

end = (len(maze) - 1, len(maze[0]) - 2)

queue = PriorityQueue()

queue.put((0, start, [start]))

visited = set()

while not queue.empty():

p, current, path = queue.get()

if current == end:

return p, path

visited.add(current)

for neighbor in neighbors(maze, current):

if neighbor not in visited:

new\_path = path + [neighbor]

priority = len(new\_path) + heuristic(neighbor, end)

queue.put((priority, neighbor, new\_path))

return None

def main():

filename = "maze-for-u.txt"

maze = read\_maze(filename)

way1 = find\_path(maze)

way2 = find\_path\_a\_star(maze)

way22 = way2[1]

for place in way1:

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

result1 = ""

for line in maze:

result1 += "".join(line) + "\n"

for place in way22:

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)

print("Сработало!")

main()