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 Панкратова Е.Д.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc136376478)

[Задачи 3](#_Toc136376479)

[1.Теоретическая часть 4](#_Toc136376480)

[а) Поиск в ширину 4](#_Toc136376481)

[б) Алгоритм А\* 6](#_Toc136376482)

[в) Структурное программирование 8](#_Toc136376483)

[2. Реализация алгоритма 10](#_Toc136376484)

[Пример работы 13](#_Toc136376485)

[Заключение 14](#_Toc136376486)

[Список литературы 15](#_Toc136376487)

[Приложение 1 16](#_Toc136376488)

[Листинг программы 16](#_Toc136376489)

# Введение

В современном информационном обществе, где данные играют все более важную роль, а требования к эффективности и оптимальности решений становятся все более строгими, разработка эффективных алгоритмов поиска пути является одной из ключевых задач. Алгоритмы поиска пути находят широкое применение в таких областях, как навигация, логистика, компьютерные игры, робототехника и другие.

Структурное программирование является подходом к разработке программного обеспечения, основанным на использовании структур данных и модульности. Оно позволяет создавать программы, которые легко понимать, поддерживать и модифицировать. Целью структурного программирования является создание программ, состоящих из небольших, логически связанных блоков кода, которые выполняют конкретные задачи.

В данной работе будут рассмотрены основные алгоритмы поиска пути, а также принципы структурного программирования. Будет изучено различные подходы к решению задач поиска пути, включая алгоритм A\*, алгоритм поиска в ширину, а также их применение.

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

# Задачи

1. Изучить алгоритмы построения маршрута в графе;
2. Подготовить исходные данные: лабиринт, координаты точек для посещения при обходе;
3. Реализовать алгоритмы с заданными параметрами;
4. Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

# 1.Теоретическая часть

## а) Поиск в ширину

Поиск в ширину (BFS) является одним из основных алгоритмов на графах. Он применяется для нахождения кратчайшего пути или определения связности между вершинами в неориентированных или ориентированных графах. В основе BFS лежит принцип поиска по слоям, начиная с заданной вершины.

Основная идея BFS заключается в том, чтобы исследовать все соседние вершины текущей вершины перед переходом к следующему слою графа. Алгоритм BFS использует очередь для хранения вершин, которые должны быть обработаны. При начале алгоритма стартовая вершина помещается в очередь, затем извлекается, и ее соседние вершины добавляются в очередь. Этот процесс повторяется до тех пор, пока очередь не опустеет, что означает исследование всех достижимых вершин.

Алгоритм BFS имеет ряд преимуществ. Во-первых, он гарантирует нахождение кратчайшего пути, то есть пути с минимальным количеством ребер. Во-вторых, BFS эффективен при поиске пути в графах с постоянным весом ребер. Также он может быть использован для решения других задач, например, определения связности графа или проверки наличия циклов.

Помимо своей широкой применимости, алгоритм BFS является простым в реализации и понимании. Он является одним из базовых алгоритмов, которые широко используются в программировании и анализе данных. Кроме того, BFS может быть оптимизирован и адаптирован для работы с различными типами графов и специфическими требованиями задачи.

В заключение, алгоритм поиска в ширину (BFS) является мощным инструментом для решения задач на графах, таких как нахождение кратчайшего пути и определение связности. Его простота и эффективность делают его важным элементом в арсенале программистов и исследователей, работающих с графами.

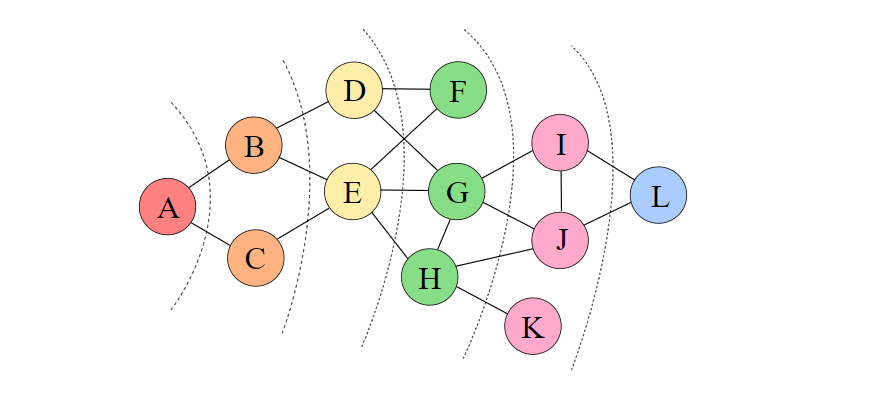


Рисунок 1 – Пример поиска в ширину на графе

## б) Алгоритм А\*

Алгоритм A\* (A-star) является одним из наиболее популярных и эффективных алгоритмов поиска пути в графах. Он используется для нахождения оптимального пути между двумя вершинами в графе, учитывая эвристическую оценку расстояния до цели. А\* комбинирует преимущества двух других алгоритмов - поиска в глубину и поиска в ширину - и является эффективным в пространстве и времени.

Основная идея алгоритма A\* заключается в том, чтобы выбирать следующую вершину для исследования, основываясь на сумме двух значений: стоимости пути от начальной вершины до текущей и эвристической оценки стоимости от текущей вершины до целевой. Эвристическая функция, обычно называемая "эвристикой", предоставляет оценку расстояния до цели и используется для выбора наилучшего пути на основе прогнозируемой информации. Часто в качестве эвристики используется расстояние по прямой (например, эвклидово расстояние) или расстояние Манхэттена.

Преимуществом алгоритма A\* является его способность находить оптимальный путь, то есть путь с минимальной стоимостью, при условии, что эвристическая функция является допустимой и не переоценивает расстояние до цели. Кроме того, алгоритм A\* обладает эффективностью по сравнению с простым поиском в ширину или поиском в глубину, поскольку он использует информацию о расстоянии до цели для ориентирования поиска в наиболее перспективных направлениях.

Однако стоит отметить, что эффективность алгоритма A\* зависит от эвристической функции, которую необходимо выбрать и оптимизировать в зависимости от конкретной задачи и графа. Некорректно выбранная эвристика может привести к неправильному результату или ненужным расходам вычислительных ресурсов.

Алгоритм A\* представляет собой мощный инструмент для нахождения оптимального пути в графах, учитывая эвристическую оценку расстояния до цели. Его преимущества включают эффективность и возможность нахождения оптимальных решений. Однако необходимо тщательно выбирать и оптимизировать эвристическую функцию для достижения наилучших результатов. Алгоритм A\* является важным инструментом в области планирования пути, игрового разработки, робототехники и других областей, где требуется эффективное нахождение пути.

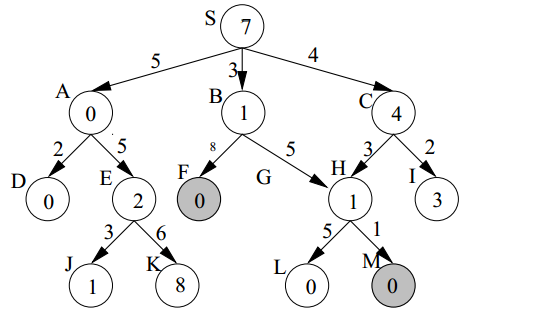


Рисунок 2 – Пример алгоритма А\* на графе

## в) Структурное программирование

Структурное программирование является подходом к разработке программного обеспечения, который стремится создать структурированный, легко понятный и легко поддерживаемый код. Этот подход подразумевает разбиение программы на небольшие, логически связанные блоки кода, называемые модулями или функциями, и организацию их в иерархическую структуру

Одной из основных целей структурного программирования является упрощение процесса разработки программ путем разделения задач на подзадачи и модули. Каждый модуль выполняет определенную функцию и является независимым и переиспользуемым компонентом. Это позволяет разработчикам концентрироваться на решении конкретных задач и повторно использовать код, что ускоряет процесс разработки и повышает его качество.

Структурное программирование также акцентирует внимание на контроле потока выполнения программы. Вместо использования сложных и запутанных конструкций, таких как безусловные переходы или переходы на случайные точки в программе, структурное программирование настаивает на использовании линейного иерархического потока управления. Это делает код более понятным и предсказуемым, что упрощает отладку и обнаружение ошибок.

Принципы структурного программирования включают в себя использование последовательных операций, условных операторов и циклов для управления потоком выполнения программы. Они также подразумевают создание четких и недвусмысленных алгоритмов, которые можно понять и проверить на корректность. Разделение ответственности и модульность являются ключевыми аспектами структурного программирования, поскольку они способствуют легкости понимания и сопровождения кода.

Структурное программирование было разработано в 1960-х годах в ответ на проблемы, связанные с использованием безусловных переходов и неструктурированным кодом. Оно стало широко принятым и применяемым подходом в программировании и обусловило появление других парадигм, таких как объектно-ориентированное программирование.

Структурное программирование представляет собой подход к разработке программного обеспечения, ориентированный на создание структурированного, понятного и легко поддерживаемого кода. Оно способствует разделению задач на подзадачи, повторному использованию кода и управлению потоком выполнения программы. Структурное программирование остается важным инструментом в разработке программного обеспечения и помогает программистам создавать качественные и эффективные программы.

# 2. Реализация алгоритма

* Описание функции read\_maze:

Для того чтобы получить лабиринт в виде двумерного массива, мы создаем функцию read\_maze. Эта функция принимает входной текстовый файл с лабиринтом и возвращает его в виде двумерного массива под названием maze. Кроме того, мы также определяем функцию get\_neighbors, которая принимает лабиринт maze в качестве аргумента и возвращает список valid\_neighbors, содержащий кортежи с координатами доступных ячеек, в которые можно переместиться. После этого мы разрабатываем алгоритм поиска в ширину.

* Описание функции BFS и get\_heighbors

Мы создаем функцию BFS, в которую входит лабиринт maze как параметр. Вначале мы указываем точки старта и конца для поиска и сохраняем их в переменных start и end соответственно.

Далее мы создаем очередь queue и добавляем в нее кортеж (start, [start]). Первый элемент в этом кортеже - это координаты начальной точки, а второй - путь к этой точке.

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

Мы проверяем каждого соседа, чтобы убедиться, что мы еще не посетили его ранее. Если он не был посещен, мы 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.

* Описание функции heuristic:

Мы создаем функцию heuristic, которая вычисляет эвристическое расстояние от текущей ячейки до конечной точки в алгоритме A\*. Входными параметрами функции являются координаты текущей ячейки cell и координаты конечной точки end. Функция использует формулу Евклида: sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2) для вычисления расстояния между текущей ячейкой и конечной точкой. Здесь cell[0] и cell[1] - это координаты текущей ячейки, а end[0] и end[1] - координаты конечной точки. (Эвристическое расстояние используется для оценки стоимости пути от текущей ячейки до конечной. 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.)

* Описание функции A\_star:

Мы создаем функцию A\_star, которая принимает лабиринт maze в качестве входного параметра. Сначала мы определяем начальную и конечную точки start и end. Затем мы создаем приоритетную очередь queue и добавляем в нее кортеж, содержащий стоимость пути, текущую точку и путь к текущей точке.

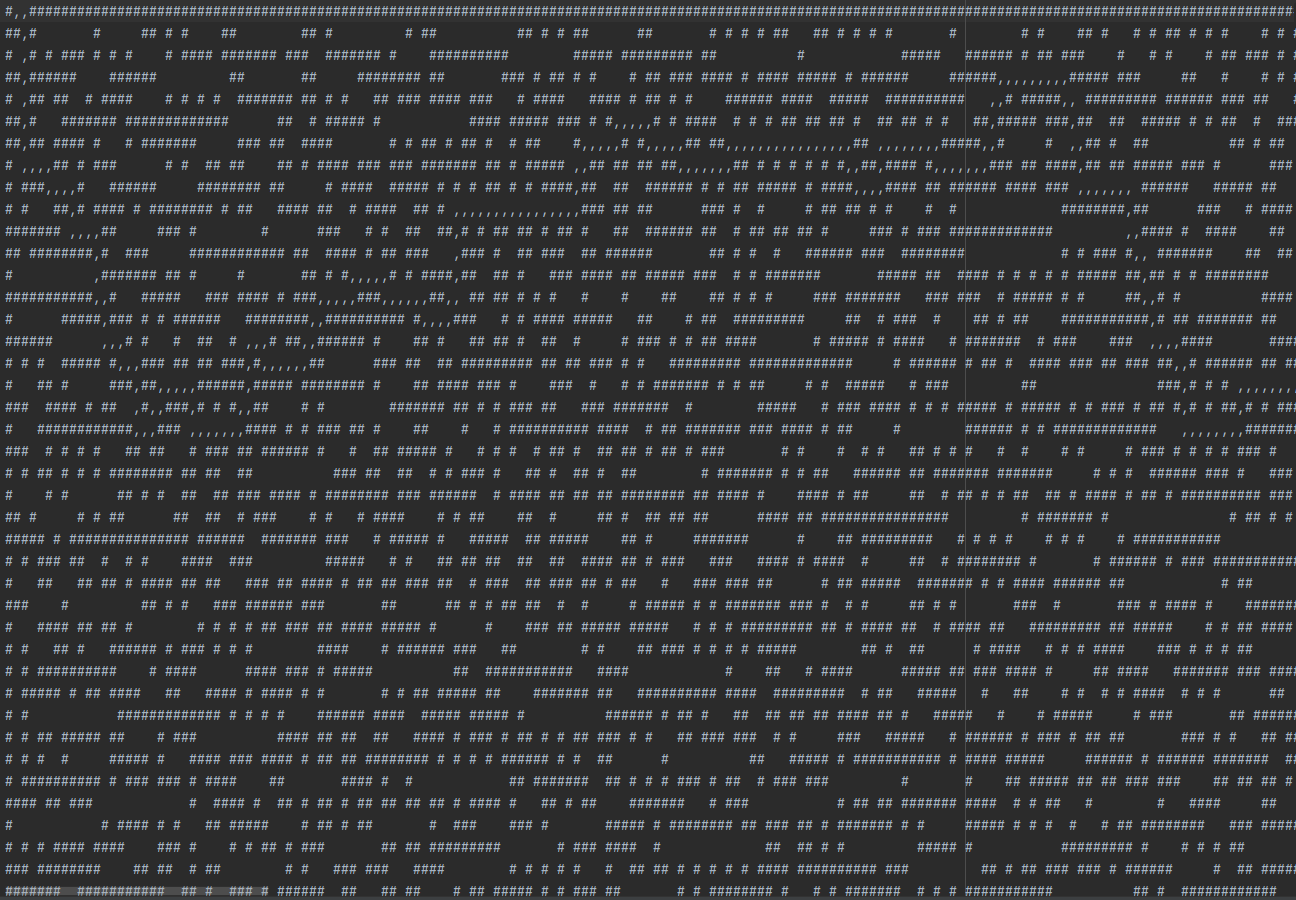
Затем мы создаем множество visited, в котором будут храниться посещенные точки, чтобы избежать повторных посещений. В цикле while, пока очередь не станет пустой, мы извлекаем кортеж с минимальной стоимостью из приоритетной очереди. Затем мы проверяем, является ли текущая точка конечной. Если это так, то функция возвращает стоимость пути и путь к конечной точке. Если текущая точка не является конечной, мы добавляем ее в множество visited, чтобы не посещать ее снова. Затем для каждого соседа текущей точки вызываем функцию get\_neighbors. Если соседняя точка еще не была посещена, мы создаем новый путь к этой точке, добавляя ее к пути к текущей точке. Затем мы вычисляем стоимость нового пути с использованием формулы priority = len(new\_path) + heuristic(neighbor, end). Здесь len(new\_path) - это фактическая стоимость пути от начальной точки до текущей, а heuristic(neighbor, end) - это эвристическое расстояние от соседней точки до конечной точки. Общая стоимость нового пути вычисляется как сумма фактической стоимости пути и эвристического расстояния. Мы создаем новый кортеж с общей стоимостью, соседней точкой и новым путем, и добавляем его в приоритетную очередь. Это продолжается до тех пор, пока конечная точка не будет достигнута или пока приоритетная очередь не станет пустой. Если конечная точка не может быть достигнута из начальной точки, функция возвращает None.

# Пример работы

Программа выполняет поиск пути, а результат работы алгоритма сохраняется в файле с названием "maze-for-me-done.txt". В этом файле представлен лабиринт с определенными символами:

1. символ "." обозначает траекторию пути, найденную алгоритмом поиска в ширину
2. символ "," обозначает траекторию пути, найденную алгоритмом А\*.

Пример работы программы:



# Заключение

В процессе выполнения данной курсовой работы мы изучили теоретические основы алгоритмов прохода по графу в ширину и А\*, а также структурного программирования.

Главное различие между А\* и поиском в ширину заключается в их стратегиях поиска. Поиск в ширину исследует все возможные пути равномерно, в то время как А\* использует информацию о стоимости пути и эвристическую оценку для более эффективного фокусирования на перспективных путях к цели.

Мы разработали программу на языке Python, которая реализует алгоритмы прохода по графу в ширину и А\* для поиска оптимального пути в лабиринте. Результаты проходов по лабиринту и найденные пути были сохранены в файле. В результате получилась программа, которая способна находить оптимальный путь в лабиринте, используя алгоритмы прохода по графу в ширину и А\*, а также сохраняющая полученные пути в файле.

# Список литературы

1. Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Структуры данных: Учебник. – Нижний Новгород: Изд-во ННГУ, 2005. 307 с. (Дата обращения 27.05.2023)
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 (Дата обращения 29.05.2023)
3. Гришутин А., Алексеев С., Иванов М., «Алгоритмика - Поиск в ширину» [Электронный ресурс]. URL: <https://ru.algorithmica.org/cs/shortest-paths/bfs/> (Дата обращения: 29.05.2023)
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 27.05.2023)
5. Лекция 10. Структурное программирование, предпрограммная подготовка задачи// StudFiles / [Электронный ресурс]. URL: <https://studfile.net/preview/4290050/> (Дата обращения: 29.05.2023)
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 29.05.2023)
7. Структурное программирование сверху-вниз и правильность программ // Allbest/ [Электронный ресурс]. URL: https://revolution.allbest.ru/programming/00712728\_0.html (Дата обращения: 29.05.2023)

# Приложение 1

## Листинг программы

import queue as q

import math as m

#Читаем лабиринт

def read\_maze(file):

with open(file) as f:

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

return maze

def get\_neighbors(maze, cell):

r, c = cell

neighbors = [(r - 1, c), (r + 1, c), (r, c - 1), (r, c + 1)]

valid\_neighbors = []

for n in neighbors:

r, c = n

if 0 <= r < len(maze) and 0 <= c < len(maze[0]) and maze[r][c] != "#":

valid\_neighbors.append(n)

return valid\_neighbors

#Поиск в ширину

def BFS(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 get\_neighbors(maze, current):

if neighbor not in visited:

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

return None

#Эвристическое расстояние

def heuristic(cell, end):

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

#A

def A\_star(maze):

start = (0, 1)

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

queue = q.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 get\_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

file = "maze-for-u.txt"

maze = read\_maze(file)

path1 = BFS(maze)

path2 = A\_star(maze)

path2 = path2[1]

for place in path1:

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

result1 = ""

for line in maze:

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

for place in path2:

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)