Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Нижегородский государственный архитектурно-строительный университет»

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,

ФИЭСИС, группы ИС-33

Горев Е.М.

Проверил: преподаватель,

Морозов Н.С.

Нижний Новгород, 2023 г.

Содержание

[Введение 3](#_Toc136200179)

[Цель работы: 3](#_Toc136200180)

[Задачи: 3](#_Toc136200181)

[1.Теоретическая часть 4](#_Toc136200182)

[1.1 Алгоритмы поиска пути 4](#_Toc136200183)

[1.1.1 Алгоритм поиска в глубину 4](#_Toc136200184)

[1.1.2 Алгоритм А\* 5](#_Toc136200185)

[1.2 Структурное программирование 8](#_Toc136200186)

[2. Реализация программы 10](#_Toc136200187)

[Пример работы 13](#_Toc136200188)

[Заключение 14](#_Toc136200189)

[Список литературы 15](#_Toc136200190)

[Приложения 16](#_Toc136200191)

# Введение

Алгоритмы поиска пути являются важной областью в информатике и компьютерных науках. Они позволяют находить оптимальные или приближенные пути между различными точками в графах или сетях. Эти алгоритмы имеют широкий спектр применений, включая навигацию роботов, планирование маршрутов, оптимизацию логистических задач и даже анализ социальных сетей.

## **Цель работы:**

Целью данной работы является исследование и реализация различных алгоритмов поиска пути на языке программирования Python. В ходе работы будут рассмотрены два алгоритма поиска пути: алгоритм поиска в глубину (depth first search, DFS) и алгоритм А\*. Данные алгоритмы будут использованы для решения задачи поиска маршрута в лабиринте.

Для достижения поставленной цели будет использовано структурное программирование в Python. Структурное программирование предлагает методологию разработки программ, основанную на использовании блоков, последовательности операций и условных конструкций. Это позволяет создавать четкие, легко читаемые и поддерживаемые программы.

## **Задачи:**

* Изучить теорию алгоритмов прохода по графу: DFS и A\*;
* Изучить парадигму структурного программирования;
* Используя исходные данные: файл с текстовым представлением лабиринта, координаты ключа, координаты начала и конца лабиринта, написать программу для поиска маршрута в лабиринте с помощью алгоритма обхода в глубину и А\*;
* Сохранить получившийся результат в файл: маршрут прохода лабиринта.

# 1.Теоретическая часть

## 1.1 Алгоритмы поиска пути

Рассмотрим алгоритмы поиска путей в графе.

### Алгоритм поиска в глубину

Поиск в глубину (англ. Depth-first search, DFS) — один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно [1, c. 169-173].

При поиске в глубину посещается первая вершина, затем необходимо идти вдоль ребер графа, до попадания в тупик. Вершина графа является тупиком, если все смежные с ней вершины уже посещены. После попадания в тупик нужно возвращаться назад вдоль пройденного пути, пока не будет обнаружена вершина, у которой есть еще не посещенная вершина, а затем необходимо двигаться в этом новом направлении. Процесс оказывается завершенным при возвращении в начальную вершину, причем все смежные с ней вершины уже должны быть посещены [2].

Таким образом, основная идея поиска в глубину – когда возможные пути по ребрам, выходящим из вершин, разветвляются, нужно сначала полностью исследовать одну ветку и только потом переходить к другим веткам (если они останутся нерассмотренными).

**Алгоритм поиска в глубину:**

Шаг 1. Всем вершинам графа присваивается значение не посещенная. Выбирается первая вершина и помечается как посещенная.

Шаг 2. Для последней помеченной как посещенная вершины выбирается смежная вершина, являющаяся первой помеченной как не посещенная, и ей присваивается значение посещенная. Если таких вершин нет, то берется предыдущая помеченная вершина.

Шаг 3. Повторить шаг 2 до тех пор, пока все вершины не будут помечены как посещенные.

Демонстрацию алгоритма можно наблюдать на рисунке 1.

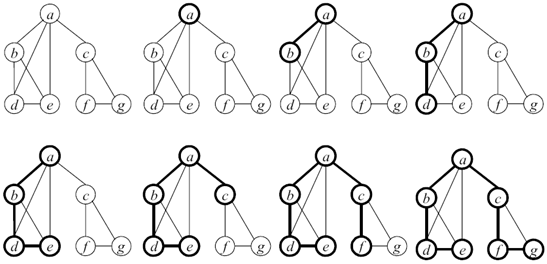


Рис. 1. Демонстрация алгоритма поиска в глубину

### 1.1.2 Алгоритм А\*

Алгоритм А\* — один из самых эффективных алгоритмов поиска кратчайшего пути между двумя точками графа. Он был опубликован Питером Хартом, Нильсом Нильссоном и Бертрамом Рафаэлем в 1968 году. Он является одним из самых часто используемых алгоритмов поиска пути.

Говоря простым языком, алгоритм А\* находит оптимальный вариант благодаря вычислению суммарной стоимости всех путей между начальной и конечной точкой. Этот способ быстрее алгоритма Дейкстры благодаря эвристической функции [3].

Алгоритм А\* характеризуется тремя важными свойствами:

оптимальностью — это означает, что алгоритм гарантирует получение лучшего из возможных результатов;

полнотой — это означает, что алгоритм А\* всегда найдет решение, если оно существует;

эффективностью — на сегодняшний день нет других алгоритмов, которые смогут найти кратчайший путь быстрее, чем А\*, применяя эвристическую функцию.

Алгоритм поиска А\* несет в себе следующую идею: изначально он всегда посещает вершины, которые, скорее всего, ведут по кратчайшему пути к цели. Такие вершины он определяет по формуле:

F(x) = G(x) + H(x), где:

F(x) — это функция для вершины; чем меньше функция, тем «ближе» вершина стоит в очереди для посещения; данная функция оценивает минимальную стоимость перехода от вершины к вершине;

G(x) — это стоимость пути от первоначальной вершины и до любой другой;

H(x) — это эвристический показатель стоимости пути от вершины «х» и до конечной вершины.

Вычисляя F(x), за «х» берется соседняя вершина с уже посещенной. Поэтому получается, что алгоритм вычисляет стоимость всего пути у всех соседних вершин и останавливает выбор на той, у которой стоимость минимальна [4].

**Принцип работы:**

Посетив одну конкретную вершину, алгоритм А\* перед переходом к следующей исследует все соседние вершины. Все вершины алгоритм разделяет на 3 категории:

Неизвестные вершины. Это те, которые не были еще посещены и пока что даже не найдены. Получается, что и путь к ним пока остается загадкой. Таким образом, изначально все вершины, кроме стартовой, будут в этой категории.

Известные вершины. Это те вершины, о которых уже известно алгоритму и уже даже известен путь к ним. Такие вершины сохраняются в «списке алгоритма» и становятся в очередь для их посещения и исследования. Из этого списка исследуются те вершины, которые считаются наиболее перспективными.

Исследованные вершины. В эту категорию попадают те вершины, которые уже были посещены алгоритмом А\*. К этим вершинам известен самый короткий путь, поэтому они попадают в «закрытый список» — этот список нужен для того, чтобы исключить многократное исследование одних и тех же вершин.

Когда одна из вершин становится полностью исследованной, она попадает в категорию «исследованные вершины», а все ее соседи попадают в категорию «известные вершины» и становятся годными для исследования. На каждой уже исследованной вершине устанавливается указатель до той уже исследованной вершины, к которой у нее будет кратчайший путь.

Алгоритм А\* завершает свою работу только в том случае, если конечная вершина переносится в категорию «исследованные вершины». В этом случае уже будет весь список исследованных вершин, а на каждой из них будет стоять указатель с кратчайшим путем. Поэтому несложно будет по указателям отследить кратчайший путь от конечной вершины до начальной.

Алгоритм А\* находит кратчайший путь между вершинами, основываясь на стоимости и «весе» ребер. Поэтому путь, который находит А\*, можно по праву назвать «самым быстрым» или «самым простым». По этой причине алгоритм А\* очень часто применяется как раз для планирования кратчайших путей, однако его также часто применяют в играх [4].

Пример работы алгоритма можно наблюдать на рисунке 2.

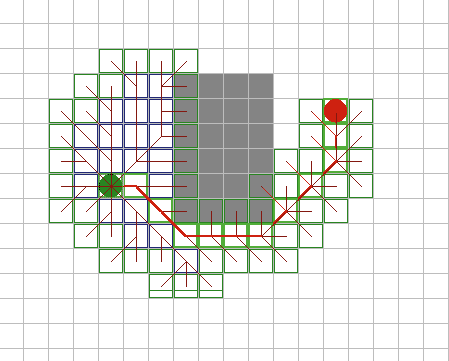


Рис. 2. Пример работы алгоритма А\*

## 1.2 Структурное программирование

Структурное программирование – технология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры логически целостных фрагментов (блоков) [5].

**Принципы структурного программирования:**

1. Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл. Их можно увидеть на рисунке 3.

2. В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом.

3. Повторяющиеся фрагменты программы можно оформить в виде подпрограмм (процедур и функций). В виде подпрограмм можно оформить логически целостные фрагменты программы, даже если они не повторяются.

4. Все перечисленные конструкции должны иметь один вход и один выход.

5. Разработка программы ведётся пошагово, методом «сверху вниз» (метод последовательной детализации).

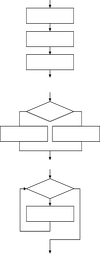


Рис. 3. Управляющие конструкции.

# 2. Реализация программы

Программа должна выполнять следующие действия: получить текстовый файл с лабиринтом, определить координаты ключа, начала и выхода лабиринта, производить поиск пути с использованием поиска в глубину и А\*, записывать в файл обновленный лабиринт с маршрутами алгоритмов.

Для каждого действия будет использоваться отдельная функция, для простоты кода и удобства его прочтения.

Рассмотрим функции, используемые в программе.

Функция read\_maze\_file открывает файл с лабиринтом, читает его построчно и создает двумерный список maze, где каждая строка представляет собой список символов. Функция возвращает полученный лабиринт.

Функция find\_start принимает лабиринт в виде двумерного списка и ищет координату начала лабиринта, то есть первую пустую клетку в верхней строке. Функция возвращает найденные координаты в формате (x, y).

Функция find\_exit принимает лабиринт в виде двумерного списка и ищет координату выхода из лабиринта, то есть последнюю пустую клетку в нижней строке. Функция возвращает найденные координаты в формате (x, y).

Функция find\_key принимает лабиринт в виде двумерного списка и ищет координату ключа в лабиринте, то есть клетку, содержащую символ '\*'. Функция возвращает найденные координаты в формате (x, y).

Функция get\_neighbors принимает лабиринт в виде двумерного списка и координаты (x, y) и возвращает список соседних координат для данной клетки. В данном случае соседние клетки ищутся только вверх, вниз, влево и вправо. Если соседняя клетка находится внутри лабиринта (не выходит за границы) и не является стеной (символом '#'), она добавляется в список соседних координат neighbors = [ ].

Функция depth\_first\_search принимает лабиринт в виде двумерного списка, координаты начала и ключа и выполняет поиск в глубину от начальной координаты до ключа в лабиринте. Она использует стек для хранения координат и множество visited для отслеживания посещенных координат. Также функция использует словарь paths для хранения пути до каждой координаты. В процессе обхода лабиринта, функция проверяет каждую координату и добавляет ее в стек, если она является соседней и еще не была посещена. Когда достигается координата ключа, функция возвращает путь до этой координаты.

Функция heuristic вычисляет эвристическое значение (приближенную стоимость) между двумя координатами. В данном случае используется расстояние, которое равно сумме абсолютных разностей между x-координатами и y-координатами.

Функция a\_star\_search принимает лабиринт в виде двумерного списка, координаты ключа и выхода и выполняет поиск пути от ключа до выхода с использованием алгоритма A\*. Она использует кучу (heap) для хранения координат с приоритетом, словарь path\_costs для хранения стоимости пути до каждой координаты и словарь paths для хранения пути до каждой координаты. В процессе обхода лабиринта, функция вычисляет стоимость перемещения к каждой соседней координате, а затем вычисляет приоритет для каждой соседней координаты, основываясь на стоимости пути и эвристическом значении. Затем координаты добавляются в кучу с приоритетом, а путь до соседней координаты обновляется в словаре paths. Когда достигается координата выхода, функция возвращает путь до этой координаты.

Функция update\_maze\_with\_path принимает лабиринт в виде двумерного списка, путь в виде списка координат и символ, и обновляет лабиринт, заменяя клетки на пути символом.

Функция save\_maze\_to\_file открывает файл, записывает измененное содержимое лабиринта в файл.

После следует основная часть программы, где данные функции применяются.

Программа считывает лабиринт из файла и сохраняет его в виде двумерного списка.

Происходит поиск начальной координаты, координаты ключа и координаты выхода в лабиринте.

Программа выполняет поиск в глубину (DFS) от начальной координаты до ключевой координаты.

Затем применяется цикл. Если путь от начальной координаты до ключевой координаты найден, программа обновляет лабиринт, заменяя ячейки вдоль пути точками, обозначающими путь.

Затем выполняется поиск по алгоритму A\* от ключевой координаты до выходной координаты. Если найден путь от ключевой координаты до выходной координаты, программа обновляет лабиринт, заменяя ячейки вдоль этого пути запятыми, обозначающими путь.

Обновленный лабиринт сохраняется в файле с именем «maze-for-me-done.txt».

# Пример работы

При запуске, программа читает лабиринт, находит путь от начальной точки к ключу, а затем от ключа к выходу, используя алгоритмы поиска DFS и A\*. Программа обновляет лабиринт обнаруженными путями и сохраняет измененный лабиринт в файл. Так же программа оповещает о найденном маршруте, как показано на рисунке

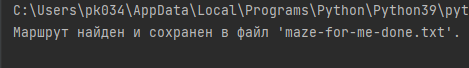


Рис. 4. Оповещение о найденном маршруте

В файле ‘maze-for-me-done.txt’ находится результат работы алгоритма, он представляет собой лабиринт, где “\*” - ключ, “#” - стена, “ ” – коридор, “.” - траектория пути алгоритма DFS, “,” - траектория пути алгоритма А\*. Пример результата работы программы изображен на рисунке 5.



Рис. 5. Результат работы программы

# Заключение

В ходе данной работы было исследовано и реализовано два алгоритма поиска пути: алгоритм поиска в глубину (DFS) и алгоритм A\*. Они являются важными инструментами в области информатики и компьютерных наук, позволяющими находить оптимальные или приближенные пути между различными точками в графах или сетях.

В процессе работы были изучены теоретические основы алгоритмов DFS и A\*, а также парадигма структурного программирования. Затем была разработана программа, которая применяет эти алгоритмы для поиска маршрута в лабиринте.

В результате работы программы был найден маршрут прохода по лабиринту, и этот результат был сохранен в файл. Реализация алгоритмов DFS и A\* дала возможность эффективно решить задачу поиска пути в лабиринте и обнаружить путь от начала до конца.

Программа успешна прошла тестирование на нескольких сформированных лабиринтах.

# Список литературы

1. Левитин А. В. Глава 5. Метод уменьшения размера задачи: Поиск в глубину // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — 576 с. — ISBN 978-5-8459-0987-9
2. Алгоритмы на графах. Алгоритмы обхода графа // intuit.ru / [Электронный ресурс]. URL: <https://intuit.ru/studies/courses/648/504/lecture/11474?page=2> (Дата обращения: 28.05.2023)
3. Алгоритм А\* и его реализация на Python // pythonist.ru / [Электронный ресурс]. URL: <https://pythonist.ru/algoritm-a-star-i-ego-realizacziya-na-python/> (Дата обращения: 28.05.2023)
4. Поисковый алгоритм A Star: что это и как эффективно его использовать? // codernet.ru / [Электронный ресурс]. URL: <https://codernet.ru/articles/web/poiskovyij_algoritm_a_star_chto_eto_i_kak_effektivno_ego_ispolzovat/> (Дата обращения: 28.05.2023)
5. Структурное программирование // skobelevserg.jimdofree.com / [Электронный ресурс]. URL: <https://clck.ru/34YCYL> (Дата обращения: 28.05.2023)
6. Кормен Т., Лейзерсон Ч., Ривест Р. Глава 22. Элементарные алгоритмы для работы с графами // Алгоритмы: построение и анализ (второе издание). — М.: «Вильямс», 2005. — 632 c.
7. Easy A \* (звезда) Поиск пути // digitrain.ru / [Электронный ресурс]. URL: <https://digitrain.ru/articles/337034/> (Дата обращения: 28.05.2023)

# Приложения

Приложение 1

Листинг программы

import heapq  
  
# Функция для чтения файла с лабиринтом и возвращения его в виде двумерного списка.  
def read\_maze\_file(file\_path):  
 maze = []  
 with open(file\_path, 'r') as file:  
 for line in file:  
 maze.append(list(line.strip()))  
 return maze  
  
# Функция для поиска начальной координаты в лабиринте.  
def find\_start(maze):  
 for y in range(len(maze[0])):  
 if maze[0][y] == ' ':  
 return (0, y)  
  
# Функция для поиска координаты выхода в лабиринте.  
def find\_exit(maze):  
 for y in range(len(maze[0])):  
 if maze[len(maze) - 1][y] == ' ':  
 return (len(maze) - 1, y)  
  
# Функция для поиска координаты ключа в лабиринте.  
def find\_key(maze):  
 for i in range(len(maze)):  
 for j in range(len(maze[i])):  
 if maze[i][j] == '\*':  
 return (i, j)  
  
# Функция для получения соседних координат от заданной координаты в лабиринте.  
def get\_neighbors(maze, x, y):  
 neighbors = []  
 directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Верх, низ, лево, право  
 for dx, dy in directions:  
 new\_x, new\_y = x + dx, y + dy  
 if 0 <= new\_x < len(maze) and 0 <= new\_y < len(maze[0]) and maze[new\_x][new\_y] != '#':  
 neighbors.append((new\_x, new\_y))  
 return neighbors  
  
# Функция для выполнения поиска в глубину от начальной координаты до ключа в лабиринте.  
def depth\_first\_search(maze, start\_coord, key\_coord):  
 stack = [start\_coord] # Стек для хранения координат  
 visited = set() # Множество посещенных координат  
 paths = {start\_coord: []} # Словарь для хранения пути до каждой координаты  
  
 while stack:  
 x, y = stack.pop() # Возвращает элемент, удаляя его из списка  
 if (x, y) == key\_coord: # Если достигнута координата ключа, возвращаем путь  
 return paths[(x, y)] + [(x, y)]  
 if (x, y) in visited: # Если координата уже была посещена, пропускаем ее  
 continue  
 visited.add((x, y))  
 neighbors = get\_neighbors(maze, x, y) # Получаем соседние координаты  
 for neighbor in neighbors:  
 if neighbor not in visited:  
 stack.append(neighbor)  
 paths[neighbor] = paths[(x, y)] + [(x, y)] # Обновляем путь до соседней координаты  
  
  
  
def heuristic(coord1, coord2):  
 x1, y1 = coord1  
 x2, y2 = coord2  
 return abs(x1 - x2) + abs(y1 - y2)  
  
def a\_star\_search(maze, key\_coord, exit\_coord):  
 heap = [(0, key\_coord)] # Куча для хранения координат с приоритетом  
 path\_costs = {key\_coord: 0} # Словарь для хранения стоимости пути до каждой координаты  
 paths = {key\_coord: []} # Словарь для хранения пути до каждой координаты  
  
 while heap:  
 cost, coord = heapq.heappop(heap)  
 if coord == exit\_coord: # Если достигнута координата выхода, возвращаем путь  
 return paths[coord] + [coord]  
 neighbors = get\_neighbors(maze, \*coord) # Получаем соседние координаты  
 for neighbor in neighbors:  
 new\_cost = path\_costs[coord] + 1 # Стоимость перемещения к соседней координате (один шаг)  
 if neighbor not in path\_costs or new\_cost < path\_costs[neighbor]:  
 path\_costs[neighbor] = new\_cost  
 priority = new\_cost + heuristic(neighbor, exit\_coord) # Приоритет = стоимость пути + эвристическое значение  
 heapq.heappush(heap, (priority, neighbor))  
 paths[neighbor] = paths[coord] + [coord] # Обновляем путь до соседней координаты  
  
  
# обновляет лабиринт, заменяя клетки на пути символом  
def update\_maze\_with\_path(maze, path, symbol):  
 for coord in path:  
 x, y = coord  
 maze[x][y] = symbol  
  
# функция для записи в файл  
def save\_maze\_to\_file(maze, file\_path):  
 with open(file\_path, 'w') as file:  
 for row in maze:  
 file.write(''.join(row) + '\n')  
  
# Основная часть программы  
maze = read\_maze\_file('maze-for-u.txt')  
start\_coord = find\_start(maze)  
key\_coord = find\_key(maze)  
exit\_cord = find\_exit(maze)  
  
# поиск пути от начальной координаты до ключа с использованием поиска в глубину  
dfs\_path = depth\_first\_search(maze, start\_coord, key\_coord)  
  
if dfs\_path:  
 # Обновляем лабиринт с путем от начальной координаты до ключа, путь точками.  
 update\_maze\_with\_path(maze, dfs\_path, '.')  
  
 # Поиск пути от ключа до выхода с использованием алгоритма A\*  
 a\_star\_path = a\_star\_search(maze, key\_coord, exit\_cord)  
  
 if a\_star\_path:  
 # Обновляем лабиринт с путем от ключа до выхода, путь запятыми  
 update\_maze\_with\_path(maze, a\_star\_path, ',')  
  
 # Обновляем символ ключа (он был затёрт)  
 x, y = key\_coord  
 maze[x][y] = '\*'  
  
 # Сохраняем лабиринт в файл  
 save\_maze\_to\_file(maze, 'maze-for-me-done.txt')  
 print("Маршрут найден и сохранен в файл 'maze-for-me-done.txt'.")  
 else:  
 print("Невозможно найти путь от ключа до выхода.")  
else:  
 print("Невозможно найти путь от начальной координаты до ключа.")