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 Буторова А.А.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Оглавление

[Введение 3](#_Toc137201489)

[Цель курсовой работы: 3](#_Toc137201490)

[Задачи: 3](#_Toc137201491)

[1. Теоретическая часть 4](#_Toc137201492)

[Структурное программирование 4](#_Toc137201493)

[Поиск в глубину 5](#_Toc137201494)

[Алгоритм А\* 6](#_Toc137201495)

[2. Реализация алгоритма в данной программе 7](#_Toc137201496)

[Реализация алгоритма поиска в глубину в данной программе 7](#_Toc137201497)

[Реализация алгоритма A\* в данной программе 7](#_Toc137201498)

[Пример работы 10](#_Toc137201499)

[Заключение 11](#_Toc137201500)

[Список литературы 12](#_Toc137201501)

[Листинг программы 13](#_Toc137201502)

# Введение

В данной курсовой работе будут рассмотрены два известных алгоритма поиска пути, которые могут быть реализованы на языке Python - Depth-First Search и A\*. Так же будет рассмотрено структурное программирование. Структурное программирование предназначено для улучшения читаемости, тестируемости и обслуживания кода программы. Depth-First Search является простым алгоритмом поиска пути, который ищет решение путем исследования каждой ветки дерева полностью, прежде чем переходить на следующую ветку. В свою очередь, A\* — это более сложный алгоритм, который использует эвристику для нахождения оптимального пути в графе между двумя узлами. При выполнении работы будут использоваться принципы структурного программирования, такие как использование функций и разделение программы на модули. В итоге, будет разработана программа, реализующая данные алгоритмы поиска пути в структурной парадигме программирования на языке Python.

Цель курсовой работы: рассмотреть алгоритмы поиска в глубину и А\* на практике, изучить их особенности, возможности и производительность, а также разработать их реализацию на языке Python для задачи поиска маршрута в лабиринте.

## Задачи:

1. Понять и изучить, что же такое структурное программирование;

2. Изучить теорию алгоритмов обхода графа: в глубину и A\*;

3. На языке программирования Python написать программу, которая будет искать маршрут в лабиринте с помощью поиска в глубину и алгоритма А\*;

4. Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

5. Разобрать работу написанного кода, объяснить принцип работы алгоритмов и структуру программы.

# 1. Теоретическая часть

## Структурное программирование

Сущность структурного подхода к разработке ИС заключается в ее декомпозиции (разбиении) на автоматизируемые функции: система разбивается на функциональные подсистемы, которые в свою очередь делятся на подфункции, подразделяемые на задачи и так далее. Процесс разбиения продолжается вплоть до конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны.

Все наиболее распространенные методологии структурного подхода базируются на ряде общих принципов:

• принцип "разделяй и властвуй" - принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;

• принцип иерархического упорядочивания - принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

• принцип абстрагирования заключается в выделении существенных аспектов системы и отвлечения от несущественных;

• принцип формализации заключается в необходимости строгого методического подхода к решению проблемы;

• принцип непротиворечивости заключается в обоснованности и согласованности элементов;

• принцип структурирования данных заключается в том, что данные должны быть структурированы и иерархически организованы.

Э. Дейкстра предложил строить программу как композицию из нескольких типов управляющих конструкций (структур), которые позволяют повысить понимаемость логики работы программы. Программирование с использованием только таких конструкций назвали структурным*.*

Э. Дейкстра был основоположником структурного программирования. Его заслуга состоит не только в разработке этой концепции, но и в ее деятельном продвижении. В 1965 году на конгрессе IFIP Дейкстра высказал мнение, что оператор goto может быть исключен из языков программирования. Однако его мнение не вызвало сильную реакцию. Тогда Дейкстра в 1968 году посвятил оператору goto отдельное письмо редактору одного из научных журналов Н. Вирту. Годом позже Дейкстра опубликовал работу, названную «Структурное программирование», что и дало название новой концепции программирования.

Для изображения передач управления в модулях используются структурные схемы программ.

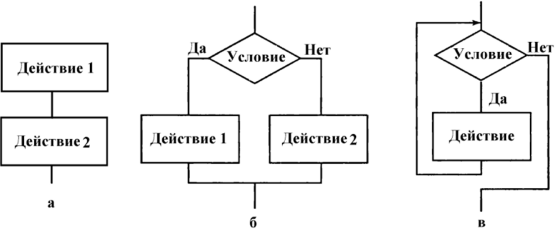


Рис 1. Структурные схемы IF-THEN-ELSE, FOR и WHILE

Основными конструкциями структурного программирования являются: следование (выполнение операторов последовательно), разветвление (в зависимости от выполнения некоторого условия выполняется та или иная последовательность операторов) и повторение (многократное выполнение одинаковой последовательности операторов). Существенно, что каждая из этих конструкций имеет по управлению только один вход и один выход. Весьма важно также, что эти конструкции являются уже математическими объектами (что, по существу, и объясняет причину успеха структурного программирования).

## Поиск в глубину

Что делать, когда приходится разгадывать лабиринт? Люди склонны выбрать маршрут и продолжать идти по нему до тех пор, пока не обнаружат тупик. Дойдя до тупика, они снова возвращаются и продолжают возвращаться, пока не увидят путь, по которому раньше не ходили. Они выберут этот новый маршрут. Еще раз продолжат идти, пока не обнаружат тупик, и вернутся снова. Именно так работает поиск по глубине.

Поиск в глубину — это рекурсивный алгоритм, который использует концепцию обратного отслеживания. Он включает в себя тщательный поиск по всем узлам, продвигаясь вперед, если это возможно, в противном случае возвращаясь назад. Здесь слово "обратный путь" означает, что как только вы продвигаетесь вперед и на текущем пути больше нет узлов, вы продвигаетесь назад по эквивалентному пути, чтобы найти узлы для прохождения. Все узлы продвигаются к посещению по текущему пути до тех пор, пока не будут пройдены все не посещённые узлы, после чего будут выбраны последующие пути.

Рекурсивный метод алгоритма поиска в глубину реализован с использованием стека. Стандартная реализация поиска в глубину помещает каждую вершину графа в одну из всех 2 категорий: Посещенная и не посещенная. Единственная цель этого алгоритма - посетить все вершины графа, избегая циклов. Алгоритм DSF выглядит следующим образом:

1. Поместите любую вершину графа на верх стека.

2. После этого возьмите верхний элемент стека и добавьте его в список посещенных вершин.

3. Затем создайте список из этого и соседнего узла вершины. Добавьте те вершины, которых нет в списке посещенных, в верхнюю часть стека.

4. Наконец, продолжайте повторять шаги 2 и 3 до тех пор, пока стек не опустеет.

## Алгоритм А\*

Алгоритм A\* (A-Star) является одним из наиболее популярных алгоритмов поиска пути в картах, используется в играх, приложениях GPS и в других областях, где важен нахождение оптимального пути.

Принцип работы алгоритма A\* основывается на использовании эвристики, чтобы выбрать более эффективный путь. Результатом работы алгоритма является наименьшее количество шагов, необходимых для достижения цели. Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как f(x)). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x) и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

Преимущество алгоритма A\* заключается в его способности находить оптимальный путь. Однако, его работу не всегда можно предсказать, особенно если карта большая, со множеством препятствий и тонкими участками. Тем не менее, алгоритм A\* остается одним из наиболее популярных алгоритмов поиска пути, используется в большом количестве приложений и игр.

# 2. Реализация алгоритма в данной программе

В первую очередь импортируется модуль heapq, который содержит функции для работы с кучей (очередью с приоритетом).

Затем определяется функция read\_maze\_file(file\_path), которая читает файл с лабиринтом и возвращает его в виде двумерного списка.

После определяем три функции, которые находят координаты входа в лабиринт (find\_start(maze)), выхода из лабиринта (find\_exit(maze)) и функция, которая находит координаты ключа (\*) в лабиринте (find\_key(maze)).

После определяется функция get\_neighbors(maze, x, y), которая получает соседние координаты от заданной координаты в лабиринте

## Реализация алгоритма поиска в глубину в данной программе

maze - двумерный список, представляющий лабиринт,

start\_coord - кортеж, представляющий начальную координату в лабиринте,

key\_coord - кортеж, представляющий координату с ключом в лабиринте.

Функция создает пустой стек stack, в котором будут храниться координаты, используемые для поиска пути. Затем создается пустое множество visited, которое будет содержать координаты, которые уже были посещены. Создается словарь paths, который хранит путь от начальной координаты до каждой координаты в лабиринте.

Затем функция начинает работу в цикле while. В этом цикле последовательно извлекается из стека одна координата x, y. Если эта координата совпадает с координатой ключа, то функция возвращает путь до этой координаты. Если же такая координата уже была посещена, то ее пропускает и переходит к следующей координате. Если координата еще не была посещена, добавляет ее во множество посещенных и получает соседние координаты с помощью функции get\_neighbors(maze, x, y). Затем функция проходит по всем соседним координатам и, если их еще не посещал, добавляет их в стек и обновляет путь в словаре paths.

Таким образом, эта функция позволяет найти путь от начальной координаты до координаты с ключом, используя поиск в глубину.

## Реализация алгоритма A\* в данной программе

Эта часть кода содержит две функции: heuristic и a\_star\_search, которые используются для поиска кратчайшего пути в лабиринте.

Функция heuristic принимает две входные координаты и вычисляет эвристическое значение между ними, используя манхэттенское расстояние. Она нужна для оценки того, насколько близки две координаты друг к другу.

Функция a\_star\_search использует алгоритм A\*, который осуществляет поиск кратчайшего пути от начальной координаты (key\_coord) до конечной координаты (exit\_coord) в лабиринте. Она создает кучу (heap), словари для хранения стоимости пути (path\_costs) и пути до каждой координаты (paths). Алгоритм работает следующим образом:

1. Из кучи извлекается координата с наименьшим значением приоритета (стоимости пути + эвристическое значение).

2. Если достигнута координата выхода (exit\_coord), то возвращаем путь до нее.

3. Получаем соседние координаты для текущей координаты.

4. Рассчитываем стоимость перемещения к соседней координате и проверяем, была ли она уже посещена (содержится ли в словаре path\_costs). Если координата не посещалась или новый путь более дешевый, то обновляем path\_costs и priority (приоритет соответствующей координаты в куче).

5. Обновляем путь до соседней координаты в словаре paths.

6. Повторяем пункты 1-5, пока куча не будет пустой.

Таким образом, функция a\_star\_search находит кратчайший путь от начальной до конечной координаты в лабиринте, используя А\* алгоритм с помощью эвристики для определения приоритета каждой координаты.

После этих алгоритмов идут:

update\_maze\_with\_path(maze, path, symbol) - функция, которая получает на вход двумерный массив maze, координату пути path и символ symbol, и заменяет символы в maze на symbol по координатам пути path.

save\_maze\_to\_file(maze, file\_path) - функция, которая получает на вход двумерный массив maze и путь к файлу file\_path, и записывает содержимое массива maze в файл file\_path.

maze = read\_maze\_file('maze-for-u.txt') - код, который считывает файл 'maze-for-u.txt' и записывает его содержимое в массив maze.

start\_coord = find\_start(maze) - код, который ищет на карте maze координату начального положения и записывает ее в переменную start\_coord.

key\_coord = find\_key(maze) - код, который ищет на карте maze координату ключа и записывает ее в переменную key\_coord.

exit\_cord = find\_exit(maze) - код, который ищет на карте maze координату выхода и записывает ее в переменную exit\_cord.

создается переменная dfs\_path, которой присваивается результат выполнения функции depth\_first\_search со следующими параметрами: лабиринт maze, начальные координаты start\_coord и координаты ключа key\_coord.

Далее, проверяется условие dfs\_path: если оно истинно, то выполняется блок кода внутри if. Если dfs\_path не содержит никаких значений (то есть False, None или пустую строку), то выполняется блок кода внутри else.

Внутри тела if вызывается функция update\_maze\_with\_path с тремя параметрами: лабиринт maze, dfs\_path (путь от начальной координаты до ключа, построенный во время выполнения функции depth\_first\_search) и символ '.'. Функция обновляет лабиринт, заменяя все клетки на пути от начальной координаты до ключа на символ '.'.

Далее, вызывается функция a\_star\_search с параметрами: лабиринт maze, координаты ключа key\_coord и координаты выхода exit\_cord для поиска кратчайшего пути от ключа до выхода с использованием алгоритма A\*.

Если функция a\_star\_search возвращает какой-то результат, выполняется блок кода внутри if. В теле блока функция update\_maze\_with\_path вызывается с параметрами: лабиринт maze, a\_star\_path (кратчайший путь от ключа до выхода, построенный во время выполнения функции a\_star\_search) и символ ','. Функция обновляет лабиринт, заменяя все клетки на пути от ключа до выхода на символ ','.

Затем, обновляется символ ключа в лабиринте, заменяя его на звездочку - символ '\*'. Это позволяет отличить клетку с ключом от других в лабиринте.

После этого вызывается функция save\_maze\_to\_file с параметрами: лабиринт maze и имя файла 'maze-for-me-done.txt'. Функция сохраняет лабиринт в файл 'maze-for-me-done.txt'.

В конце блока if выводится на экран сообщение "Маршрут найден". Если функция a\_star\_search не возвращает результат, то выполняется блок кода внутри else, который выводит сообщение "Невозможно найти путь от ключа до выхода.". Если переменная dfs\_path не содержит какого-либо значения, то выводится сообщение "Невозможно найти путь от начальной координаты до ключа.".

# Пример работы

Изображение выглядит как снимок экрана, шаблон, прямоугольный, Симметрия

Автоматически созданное описание

Рис. 2 Поиск в глубину

Изображение выглядит как снимок экрана, шаблон

Автоматически созданное описание

Рис. 3 Алгоритм А\*

# Заключение

В рамках выполнения задачи были разработан программный код для решения задачи прохождения маршрута в лабиринте. Для этого были использованы алгоритмы поиска в глубину и A\*. Алгоритм А\* и алгоритм поиска в глубину являются важными инструментами в области искусственного интеллекта и компьютерных наук. Оба алгоритма используются для решения задач поиска оптимального пути в графах и имеют свои преимущества и недостатки. Алгоритм А\* обладает более высокой эффективностью и точностью, но требует большего объема вычислений. Алгоритм поиска в глубину, в свою очередь, более прост в реализации и требует меньше вычислительных ресурсов, но может не дать оптимального решения. В результате работы был создан код для прохождения маршрута от начальной координаты аватара до ключа с помощью алгоритма поиска в глубину, а затем - от ключа до ближайшего выхода с помощью A\*.

# Список литературы

1. Авачева Т. Г., Пруцков А. В. Современный взгляд на концепцию структурного программирования // Cloud of Science. — 2019. — Т. 6. — № 4. — С. 646–665.

2. Структурное программирование // bstudy.net / [Электронный ресурс]. URL: <https://bstudy.net/915232/tehnika/strukturnoe_programmirovanie> (дата обращения: 04.05.2023)

3. Харт П., Нилсон Н., Раффо Д. Эффективный алгоритм поиска кратчайшего пути // Искусственный интеллект. – 1968. – Т. 1. – № 3. – С. 189-208.

4. A\* // Википедиа / [Электронный ресурс]URL: [https://ru.wikipedia.org/wiki/A\*](https://ru.wikipedia.org/wiki/A*) (Дата обращения: 05.05.2023)

5. Depth First Search in Python (with Code) | DFS Algorithm [Электронный ресурс] / URL: https://favtutor.com/blogs/depth-first-search-python (дата обращения: 07.05.2023).

6. Easy A \* (звезда) Поиск пути // digitrain.ru / [Электронный ресурс]. URL:<https://digitrain.ru/articles/337034/> (Дата обращения: 05.05.2023)

7. vuzlit.com [Электронный ресурс] / URL: <https://vuzlit.com/957009/poisk_glubinu> (дата обращения: 04.05.2023)

# Листинг программы

import heapq  
  
  
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)  
  
  
def find\_exit(maze):  
 for y in range(len(maze[0])):  
 if maze[len(maze) - 1][y] == ' ':  
 return (len(maze) - 1, y)  
  
  
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]  
 visited = set()  
 paths = {start\_coord: []}  
  
 while stack:  
 x, y = stack.pop()  
 if (x, y) == key\_coord:  
 return paths[(x, y)] + [(x, y)]  
 if (x, y) in visited:  
 continue  
 visited.add((x, y))  
 neighbors = get\_neighbors(maze, x, y)  
 for neighbor in neighbors:  
 if neighbor not in visited:  
 stack.append(neighbor)  
 paths[neighbor] = paths[(x, y)] + [(x, y)]  
  
  
  
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)]  
 path\_costs = {key\_coord: 0}  
 paths = {key\_coord: []}  
  
 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)  
 for neighbor in neighbors:  
 new\_cost = path\_costs[coord] + 1  
 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]  
  
  
  
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:  
 update\_maze\_with\_path(maze, dfs\_path, '.')  
 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, ',')  
  
 x, y = key\_coord  
 maze[x][y] = '\*'  
  
 # Сохраняем лабиринт в файл  
 save\_maze\_to\_file(maze, 'maze-for-me-done.txt')  
 print("Маршрут найден")  
 else:  
 print("Невозможно найти путь от ключа до выхода.")  
else:  
 print("Невозможно найти путь от начальной координаты до ключа.")