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 Пилкина А.Д.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

**Оглавление**

[**Введение** 2](#_Toc136748876)

[**Цель работы** 2](#_Toc136748877)

[**Задачи** 2](#_Toc136748878)

[**Теоретические аспекты структурного программирования** 3](#_Toc136748879)

[**Основные принципы и концепции структурного программирования** 4](#_Toc136748880)

[**Структуры данных и их применение в программировании** 6](#_Toc136748881)

[**Алгоритмы поиска пути** 8](#_Toc136748882)

[**Алгоритм А\*** 9](#_Toc136748883)

[**Жадный алгоритм** 10](#_Toc136748884)

[**Практическая часть** 11](#_Toc136748885)

[**Реализация алгоритма** 11](#_Toc136748886)

[**Пример работы алгоритма** 15](#_Toc136748887)

[**Заключение** 16](#_Toc136748888)

[**Список литературы** 17](#_Toc136748889)

[**Приложение** 18](#_Toc136748890)

[**Листинг программы** 18](#_Toc136748891)

# **Введение**

В настоящее время, с увеличением количества данных и сложности задач, появляется необходимость в разработке эффективных алгоритмов и структур данных для их обработки. Алгоритмы поиска пути и структурное программирование являются ключевыми компонентами в различных областях, таких как компьютерные игры, робототехника, геоинформационные системы и многие другие. В данной курсовой работе рассматриваются два основных алгоритма поиска пути - жадный алгоритм и алгоритм A\*. Результатом будет является программа с алгоритмами поиска пути на языке программирования Python с использованием структур данных.

## **Цель работы**

Цель данной курсовой работы состоит в изучении и реализиции алгоритмов поиска пути и структурного программирования на языке программирования Python. В работе будут рассмотрены основные алгоритмы поиска пути, такие как жадный алгоритм, алгоритм A\*.

## **Задачи**

1. Изучить теоретические аспекты структурного программирования, включая основные принципы и концепции, структуры данных и принципы построения алгоритмов.

2. Изучить алгоритмы, включая алгоритм A\* и жадный алгоритм, поиска пути на графах.

3. Разработать программный код на языке Python на основе изученных алгоритмов поиска пути для поиска маршрута в лабиринте, используя структурное программирование и структуры данных.

4. Протестировать и отладить программный код, а также пояснить принцип его работы.

**Основная часть**

## **Теоретические аспекты структурного программирования**

Структурное программирование - это методология программирования, которая была разработана в 1960-х годах. Она основана на использовании структурных блоков (таких как последовательность, ветвление и циклы), которые образуют иерархическую структуру программы. Основная идея структурного программирования заключается в том, что программа должна быть разбита на небольшие блоки, каждый из которых выполняет определенную задачу, и эти блоки должны соединяться в определенном порядке, чтобы получить полную программу [1].

Структурное программирование стало основой для различных методологий программирования, таких как объектно-ориентированное программирование и функциональное программирование [1].

### **Основные принципы и концепции структурного программирования**

Структурное программирование предполагает использование структурных блоков, таких как последовательность, ветвление и циклы. Эти блоки позволяют описывать логику программы и разбивать ее на более мелкие и понятные части.

Декомпозиция - это процесс разбиения программы на более мелкие и понятные части. Это позволяет упростить программу и облегчить ее понимание и сопровождение. Декомпозиция может быть выполнена на разных уровнях, начиная от общей структуры программы до отдельных функций и процедур.

Не мало важным является использование явных и понятных структурных блоков, которые позволяют программистам легко понимать логику программы и ее структуру. Это также позволяет упростить отладку и сопровождение программы [2].

Структурное программирование также предполагает использование модульной структуры программы. Модули - это независимые части программы, которые могут быть использованы в других проектах. Это позволяет сократить время разработки и упростить сопровождение программы. Тестирование позволяет выявлять ошибки и дефекты программы и улучшать ее качество.

В целом, структурное программирование позволяет программистам легко понимать логику программы и ее структуру, что облегчает ее сопровождение и доработку [1].

Хотя оно имеет множество преимуществ, оно также имеет некоторые недостатки.

Недостаток структурного программирования заключается в том, что оно может ограничивать свободу программистов при разработке сложных программ. Сложные программы могут требовать использования более сложных структур, которые не всегда могут быть выражены в данных рамках. Кроме того, оно может быть неэффективным в некоторых ситуациях, например, когда требуется обработка больших объемов данных или, когда необходимо реализовать сложные алгоритмы [2].

Однако, несмотря на эти недостатки, структурное программирование остается одним из наиболее популярных и эффективных подходов к программированию.

### **Структуры данных и их применение в программировании**

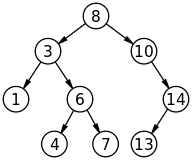
Структуры данных - это способ организации и хранения данных, который обеспечивает эффективный доступ и возможность редактирования. Они играют важную роль в программировании, поскольку позволяют эффективно хранить и обрабатывать большие объемы данных [3].

Существует множество различных структур данных, каждая из которых предназначена для определенных типов данных и задач. Некоторые из наиболее распространенных структур данных включают в себя:

- Массивы: это упорядоченная коллекция элементов, которые могут быть одного типа данных. Они могут использоваться для хранения данных, таких как числа, строки или объекты.

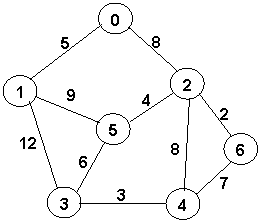
- Списки: это упорядоченная коллекция элементов, которые могут быть различных типов данных. Они могут использоваться для хранения данных, таких как имена, адреса или другие свойства.

- Деревья: это структура данных, которая использует иерархическую организацию элементов, где каждый элемент имеет один или несколько элементов-потомков. Они могут использоваться для хранения и обработки данных, таких как иерархические структуры, например, файловые системы или структуры данных, такие как бинарные деревья поиска. (Рис.1)



*Рисунок 1.*

- Графы: это структура данных, которая использует набор вершин и ребер для представления отношений между элементами. Они могут использоваться для решения задач, таких как поиск кратчайшего пути или определение связей между элементами. (Рис.2)



*Рисунок 2.*

Применение структур данных в программировании может быть разным, в зависимости от конкретной задачи. Они могут использоваться для хранения и обработки данных, поиска элементов в больших наборах данных, сортировки элементов, обхода деревьев и графов, реализации алгоритмов поиска пути и многого другого [3].

## **Алгоритмы поиска пути**

Алгоритмы поиска пути используются для нахождения оптимального маршрута между двумя точками в заданной области, такой как карта или сетка. Существует множество различных алгоритмов, каждый из которых имеет свои преимущества и недостатки в зависимости от конкретной задачи и условий, таких как размер области поиска, количество препятствий, требуемая точность и эффективность [3].

Мы рассмотрим два из них, это жадный алгоритм и алгоритм А\*.

### **Алгоритм А\***

Наилучшим алгоритмом для поиска оптимальных путей в различных пространствах является A\*.

Принцип работы алгоритма A\* основывается на использовании эвристики, чтобы выбрать более эффективный путь. Результатом работы алгоритма является наименьшее количество шагов, необходимых для достижения цели [5].

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдет минимальный путь. Как и все эвристические алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От «жадного алгоритма», который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь [7].

Составляющая g(x) — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме. В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение f(x), после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в очереди с приоритетами. Приоритет пути определяется по значению f(x) = g(x) + h(x). Алгоритм продолжает свою работу до тех пор, пока значение f(x) целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью [6].

Алгоритм хорошо подходит для поиска пути на карте или в игре, где необходимо найти оптимальный путь между двумя точками, учитывая препятствия или другие ограничения. Он может быть использован для поиска маршрутов в GPS-навигации, робототехнике, автоматическом пилотировании и других областях.

### **Жадный алгоритм**

Жадные алгоритмы — это алгоритмы, которые, на каждом шагу принимают локально оптимальное решение, не заботясь о том, что будет дальше. Они не всегда верны, но есть задачи, где жадные алгоритмы работают правильно.

Они не смотрят в будущее, чтобы выбрать глобальное оптимальное решение. Их интересует только лучшее решение в данный момент. Но общее оптимальное решение может отличаться от решения, которое выбирает алгоритм на каждом шаге своей работы. Так же они никогда не оглядываются назад на то, что сделали, чтобы понять, нужна ли глобальная оптимизация [5].

Жадные алгоритмы очень быстрые. Намного быстрее, чем две другие альтернативы. Они популярные, потому что они быстрые.

Данный алгоритм для поиска путей может использоваться в различных областях, таких как GPS-навигация, автоматический пилотирование, робототехника и других. Он может быть полезен в случаях, когда необходимо быстро найти путь к конечной точке и оптимальность пути не является критически важной. Однако, если требуется найти оптимальный путь, то жадный алгоритм может не дать правильного решения и более сложные алгоритмы, такие как A\*, могут быть более подходящими [4].

# **Практическая часть**

## **Реализация алгоритма**

1. Для начала создаем функцию read\_maze(), которая считывает лабиринт из файла с именем filename и возвращает его в виде списка списков. Для открытия файла используем конструкцию with open() as f, которая считывает его содержимое в список maze, где каждый элемент списка представляет собой строку лабиринта, преобразованную в список символов.

2. Далее для определения высоты и ширины лабиринта, используем функцию len(). Затем программа ищет все проходы в лабиринте и добавляет их в список passages. Из этого списка выбирается случайная клетка, в которой будет находиться ключ.

3. Функция get\_neighbors() принимает на вход лабиринт maze и клетку cell в виде кортежа (row, col). Она определяет соседей клетки cell, проверяя, находятся ли они внутри лабиринта и не являются ли стенами. Валидные соседи добавляются в список valid\_neighbors, который затем возвращается функцией.

4. Функция get\_heuristic() нужна, чтобы вычислять эвристическое расстояние от ячейки до ключа или конечной точки, 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. Результат вычислений возвращается функцией.

5. Жадный алгоритм

1. Определение стартовой точки start и точки с ключом key.

2. Создание стека stack, содержащего кортежи (current, path), где current - текущая точка, а path - путь до текущей точки.

3. Создание множества visited, содержащего посещенные точки.

4. Пока стек stack не пуст, извлекаем кортеж (current, path) из стека.

5. Если current является точкой с ключом, возвращаем путь path.

6. Добавляем current в множество visited.

7. Получаем список валидных соседей neighbors для текущей точки с помощью функции get\_neighbors().

8. Сортируем список neighbors по возрастанию эвристического расстояния до точки с ключом с помощью лямбда-функции.

9. Для каждого соседа neighbor из списка neighbors проверяем, не посещали ли мы его ранее. Если нет, добавляем кортеж (neighbor, path + [neighbor]), где neighbor — это координаты соседней точки, а path + [neighbor] — это путь до неё, в стек stack.

10. Если путь до точки с ключом не найден, возвращаем None.

6. Алгоритм А\*

Программа find\_path\_a\_star использует алгоритм A\* для поиска кратчайшего пути от случайно сгенерированного ключа до конца лабиринта.

1. Функция сначала устанавливает переменную key в глобальную переменную random\_key, которая является кортежем из двух целых чисел, представляющих координаты ключа в лабиринте.

2. Переменная end устанавливается в кортеж из двух целых чисел, представляющих координаты конца лабиринта. Конец предполагается вторым с конца элементом в последнем ряду лабиринта.

3. Инициализируется очередь с приоритетом с помощью класса PriorityQueue из модуля queue. Очередь будет хранить кортежи, содержащие приоритет, текущую ячейку и путь, пройденный для достижения этой ячейки.

4. Начальная точка поиска добавляется в очередь с приоритетом 0. Текущая ячейка устанавливается в key, а путь устанавливается в список, содержащий только key.

5. Создается пустое множество visited, которое будет использоваться для отслеживания посещенных ячеек.

6. Основной цикл функции работает, пока очередь с приоритетом не пуста.

7. В каждой итерации цикла ячейка с наибольшим приоритетом удаляется из очереди. Приоритет определяется суммой длины пройденного пути до ячейки и оценочного расстояния до конца лабиринта, рассчитанного с помощью функции get\_heuristic.

8. Если текущая ячейка является концом лабиринта, функция возвращает длину пути и сам путь, в противном случае текущая ячейка добавляется в множество visited.

10. Вызывается функция get\_neighbors, чтобы найти все допустимые соседние ячейки текущей ячейки.

11. Для каждой соседней ячейки, которая еще не была посещена, создается новый путь, добавляя соседа в конец текущего пути: new\_path = path + [neighbor].

12. Приоритет нового пути рассчитывается путем добавления длины пути к соседу до оценочного расстояния от соседа до конца лабиринта: priority = len(new\_path) + get\_heuristic(neighbor, end).

13. Сосед, его новый путь и его приоритет добавляются в очередь с приоритетом.

14. Если путь до конца лабиринта не найден, функция возвращает None.

7. В функции main() происходит чтение лабиринта из файла maze-for-u.txt с помощью функции read\_maze(), которая возвращает двумерный список, представляющий лабиринт.

Далее вызываем функции find\_path(maze) и find\_path\_a\_star(maze), которые ищут путь через лабиринт. Результаты этих функций сохраняются в переменных path1 и path2. Из переменной path2 извлекается только путь, который сохраняется в переменной path22.

В цикле for по всем точкам в path1, соответствующие ячейки в лабиринте заменяем на символ «.», также в path22 на символ «,».

Затем, результат изменений сохраняем в двух строках res1 и res2.

Результат записывается в файл rezultat.txt с помощью контекстного менеджера with open() as f: и функции записи f.write().

В конце функции main() вызывается, чтобы запустить программу.

## **Пример работы алгоритма**

Файл «rezultat.txt», получившийся в результате работы алгоритма, представляет собой лабиринт, где «#» - стена, а пустое пространство - коридор.

«.» - траектория пути жадного алгоритма, «,» - траектория пути алгоритма А\*, «\*» - ключ. В качестве примера рассмотрим небольшой участок с ключом.(Рис.3)



*Рисунок 3. Участок итогового лабиринта*

# **Заключение**

В заключении можно отметить, что изучение алгоритмов поиска пути и структурного программирования является важным для разработки программных продуктов и решения различных задач в области информационных технологий.

В работе были рассмотрены различные алгоритмы поиска пути, такие как жадный и алгоритм A\*. Было показано, как эти алгоритмы могут использоваться для решения задач поиска кратчайшего пути в лабиринте.

Также рассмотрели структурное программирование и его основные принципы, такие как использование для хранения и обработки данных, поиска элементов в больших наборах данных, сортировки элементов, обхода лабиринта, реализации алгоритмов поиска пути и многого другого. На примере написания программы мы показали данные принципы.

В целом, цель курсовой работы была достигнута. В результате мы получили программу, которая может прокладывать маршруты в лабиринте и находить кратчайший путь с сохранением результата в файл.

# **Список литературы**

1. Программирование на языке Python. Основы структурного программирования: учебное пособие для вузов / Майков К. А., Пылькин А. Н., Соколова Ю. С. [и др.]. – 2021. – 123 с. (Дата обращения: 27.05.2023)
2. Авачева Т. Г., Пруцков А. В. Современный взгляд на концепцию структурного программирования // Cloud of Science. — 2019. (Дата обращения 27.05.2023)
3. Совершенный алгоритм. Графовые алгоритмы и структуры данных. / Рафгарден Тим - Текст: электронный - 2019. URL:

<https://dokumen.pub/1nbsped-978-5-4461-1272-2.html> (Дата обращения 28.05.2023)

1. Жадные алгоритмы// Habr / [Электронный ресурс]. URL: https://habr.com/en/articles/120343/ (Дата обращения: 24.05.2023)
2. Алгоритмы на графах. Алгоритмы нахождения кратчайшего пути// ИНТУИТ национальный открытый университет/ [Электронный ресурс]. URL: <https://intuit.ru/studies/courses/648/504/lecture/11475?page=2> (Дата обращения: 24.05.2023)
3. Алгоритм поиска пути. Алгоритм А\*. Обход препятствий. / Bryan Stout, -1997. Перевод Maxim Kamensky. – 2000. / [Электронный ресурс]. URL: <http://pmg.org.ru/ai/stout.htm>
4. Эвристики для поиска кратчайших путей: статья / Викиконспекты. -2016. / [Электронный ресурс]. URL:

<https://neerc.ifmo.ru/wiki/index.php?title=%D0%AD%D0%B2%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B8_%D0%B4%D0%BB%D1%8F_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0_%D0%BA%D1%80%D0%B0%D1%82%D1%87%D0%B0%D0%B9%D1%88%D0%B8%D1%85_%D0%BF%D1%83%D1%82%D0%B5%D0%B9> (Дата обращения: 24.05.2023)

# **Приложение**

## **Листинг программы**

from queue import PriorityQueue

from math import sqrt

import random

def read\_maze(filename):

with open(filename) as f:

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

return maze

height = len(read\_maze("maze-for-u.txt"))

width = len(read\_maze("maze-for-u.txt")[0])

passages = []

for i in range(height):

for j in range(width):

if read\_maze("maze-for-u.txt")[i][j] == " ":

passages.append((i, j))

random\_key = random.choice(passages)

def get\_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

def get\_heuristic(cell, end):

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

# Жадный алгоритм

def find\_path(maze):

start = (0, 1)

key = random\_key

stack = [(start, [start])]

visited = set()

while stack:

current, path = stack.pop()

if current == key:

return path

visited.add(current)

neighbors = get\_neighbors(maze, current)

neighbors.sort(key=lambda neighbor: get\_heuristic(neighbor, key))

for neighbor in neighbors:

if neighbor not in visited:

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

return None

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

key = random\_key

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

queue = PriorityQueue()

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

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) + get\_heuristic(neighbor, end)

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

return None

def main():

filename = "maze-for-u.txt"

maze = read\_maze(filename)

path1 = find\_path(maze)

path2 = find\_path\_a\_star(maze)

path22 = path2[1]

for place in path1:

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

res1 = ""

for line in maze:

res1 += "".join(line) + "\n"

for place in path22:

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

res2 = ""

for line in maze:

res2 += "".join(line) + "\n"

with open("rezultat.txt", "w") as f:

f.write(res2)

main()