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: Боницкий Е.Д.

Проверил: Морозов Н.С.

Нижний Новгород, 2023 г.

**Содержание**

[Введение 3](#_Toc136804168)

[Цель работы: исследование 3](#_Toc136804169)

[Задачи: 3](#_Toc136804170)

[Основная часть 4](#_Toc136804171)

[Структурное программирование 4](#_Toc136804172)

[Алгоритмы поиска пути 5](#_Toc136804173)

[Алгоритм А\* 5](#_Toc136804174)

[Жадный алгоритм 5](#_Toc136804175)

[Реализация алгоритма 6](#_Toc136804176)

[Пример работы алгоритма 9](#_Toc136804177)

[Список литературы 11](#_Toc136804178)

[Приложения 12](#_Toc136804179)

# Введение

В работе рассмотрены элементы программирования, имеющие немалое значение: алгоритмы поиска пути и структурное программирование. Алгоритмы поиска пути существенны в областях, где необходимо находить оптимальные решения. Структурное программирование предназначено для улучшения читаемости, тестируемости и обслуживания кода программы. Эта работа охватит такие алгоритмы поиска пути, как жадный алгоритм и алгоритм A\*.

Цель работы: исследование и реализация алгоритмов поиска пути: жадный и А\* для задачи поиска маршрута в лабиринте.

## Задачи:

1. Изучить, что такое структурное программирование;
2. Изучить теорию алгоритмов поиска пути: жадного и А\*;
3. Написать программу на языке Python для поиска маршрута в лабиринте с помощью жадного алгоритма и алгоритма А\*.
4. Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

# Основная часть

## Структурное программирование

Структурное программирование — это метод проектирования и написания компьютерных программ, в котором используются только некоторые структуры языка программирования, такие как условия, циклы и подпрограммы, а также структуры данных.

Структурное программирование является основой для других методов программирования, таких как объектно-ориентированное программирование и функциональное программирование [6].

Структурное программирование было разработано в 60-х годах 20 века как реакция на проблемы с производительностью и надежностью программ, написанных без использования явно выраженной структуры [5; c. 448].

Главным принципом структурного программирования является тот факт, что любая программа может быть разложена на более мелкие модули. Это значительно упрощает процесс программирования, улучшить производительность и повысить надежность создаваемого программного продукта.

Некоторые из наиболее распространенных структур языка программирования включают в себя:

- Подпрограммы - позволяют избежать дублирования кода путем выделения общих операций в отдельные функции и процедуры.

- Условия - позволяют контролировать, выполняется ли определенный фрагмент кода в зависимости от того, выполняются ли заданные условия.

- Циклы - позволяют выполнять фрагменты кода многократно в соответствии с заданными условиями.

Недостатком структурного программирования является ограниченность в решении некоторых задач, которые требуют более сложных алгоритмов [1].

## Алгоритмы поиска пути

### 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 [8].

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)) [7].

Преимущество алгоритма A\* заключается в его способности находить наиболее оптимальный путь. Однако, его работу не всегда можно предсказать, особенно если карта большая, со множеством препятствий и тонкими участками [2].

Тем не менее, алгоритм A\* остается одним из наиболее популярных алгоритмов поиска пути, используется в большом количестве приложений и игр.

### Жадный алгоритм

Жадный алгоритм (Greedy algorithm) — это метод решения задач, который всегда выбирает локально наилучшее решение на каждом этапе, надеясь, что в итоге получится глобально оптимальное решение. Жадные алгоритмы используются для решения многих задач, например: минимальное остовное дерево, минимальный путь в лабиринте, покрытие множества, оптимизация расписания и т.д. [4; c. 442-443].

Основной принцип жадного алгоритма заключается в том, что на каждом этапе алгоритма выбирается наиболее оптимальный вариант, который не противоречит ограничениям задачи. Это может привести к тому, что весь алгоритм найдет оптимальное решение для задачи [3].

# Реализация алгоритма

В происходит:

Чтение лабиринта из текстового файла "maze-for-u.txt" с помощью функции read\_maze. В результате получается высота h, ширина w и двумерный список maze, представляющий структуру лабиринта.

Пользователю предлагается ввести координаты начальной позиции avatar\_x и avatar\_y, координаты ключа key\_x и key\_y, а также координаты выхода exit\_x и exit\_y в соответствии с размерами лабиринта.

Формируется кортеж начальной позиции avatar\_pos и кортеж искомой позиции ключа key\_pos, используя введенные пользователем координаты.

Запускается функция greedy(maze, avatar\_pos, key\_pos), которая ищет кратчайший путь от начальной позиции до позиции ключа в лабиринте.

Запускается функция astar(maze, key\_pos, exit\_pos), которая ищет кратчайший путь от позиции ключа до позиции выхода в лабиринте.

Запускается функция write\_maze(output\_file\_name, h, w, maze, key\_pos, path\_key, path\_exit), которая записывает новый лабиринт и маршруты от начальной позиции до ключа и от ключа до выхода в текстовый файл "maze-for-me-done.txt".

**get\_neighbors(point, maze):**

Данная функция принимает два аргумента - координаты точки (point) и двумерный массив maze, представляющий множество точек, которые являются доступными для перемещения.

Функция возвращает список соседних точек, которые можно достичь из заданной точки. Для этого проверяются соседние точки наличием стенок (проверяется индекс массива) и добавляются в список neighbors\_list, если они доступны для перемещения.

**manhattan\_distance(point1, point2):**

Данная функция вычисляет манхэттенское расстояние между двумя точками на плоскости.

Первый аргумент функции (point1) — это кортеж из двух чисел, представляющих координаты первой точки на плоскости.

Второй аргумент (point2) - также кортеж из двух чисел, представляющих координаты второй точки на плоскости.

Функция возвращает сумму модулей разностей координат x и y точек point1 и point2.

**path(came\_from, current):**

Данная функция строит путь из стартовой точки к конечной на основе словаря came\_from, который хранит информацию о том, откуда пришли в каждую точку, и конечной точки current.

Функция начинает с добавления конечной точки в список total\_path. Затем, пока текущая точка находится в словаре came\_from, функция переходит к ее родительской точке и добавляет ее в начало списка total\_path. Это повторяется, пока не будет достигнута стартовая точка, от которой не было родительской точки, связанной с ней в словаре came\_from.

Функция возвращает список точек total\_path, которые образуют путь от стартовой точки к конечной.

**greedy(maze, start, end):**

Данная функция реализует алгоритм жадного (greedy) поиска на основе эвристической функции - расстояния Манхэттена (manhattan\_distance). Алгоритм ищет кратчайший путь из стартовой точки start к конечной точке end в лабиринте maze.

Функция начинает с создания списка open\_list, содержащего кортежи из текущей точки и расстояния до нее (начальное значение расстояния равно 0). Также создаются множество closed\_list для хранения посещенных точек и словарь came\_from для хранения информации о точках, из которых пришли в текущую точку.

Цикл while выполняется до тех пор, пока open\_list не пуст. На каждой итерации цикла извлекается первый элемент из списка, то есть точка с наименьшим приоритетом (расстоянием Манхэттена). Если эта точка равна конечной точке end, то функция возвращает кратчайший путь с помощью функции path.

Если текущая точка не является конечной и не может быть достигнута из стартовой точки, то она добавляется в множество closed\_list.

Далее функция обрабатывает соседние точки, которые еще не были посещены и не являются стенами. Если такое соседнее положение еще не находится в open\_list, тогда текущей точке назначается предшественник в came\_from, и точка с приоритетом, равным расстоянию Манхэттена от нее до конечной точки end, помещается в список open\_list.

На последней строке функция возвращает значение None, если кратчайший путь не найден.

**astar(maze, start, end):**

Данная функция реализует алгоритм A\* поиска на основе эвристической функции - расстояния Манхэттена (manhattan\_distance). Алгоритм ищет кратчайший путь из стартовой точки start к конечной точке end в лабиринте maze.

Функция начинает с создания списка open\_list, содержащего начальную точку start. Также создаются пустые списки closed\_list для хранения посещенных точек, словарь g\_scores для хранения длин путей от стартовой точки до текущей и словарь f\_scores для хранения оценок стоимости оставшегося пути от текущей точки до конечной.

Цикл while выполняется до тех пор, пока open\_list не пуст. На каждой итерации цикла выбирается точка current с минимальным значением f\_scores[current]. Если эта точка равна конечной точке end, то функция возвращает кратчайший путь с помощью функции path.

Если текущая точка не является конечной и не может быть достигнута из стартовой точки, то она добавляется в список closed\_list.

Далее функция обрабатывает соседние точки, которые еще не были посещены и не являются стенами. Если такое соседнее положение еще не находится в open\_list, то для этой точки вычисляются оценки g\_scores и f\_scores, предшественник точки указывается в came\_from, и точка добавляется в open\_list. Если же точка уже имеется в open\_list, то ее оценки g\_scores и f\_scores пересчитываются, если предложенный новый путь более короткий, чем предыдущий.

На последней строке функция возвращает значение None, если кратчайший путь не найден.

**read\_maze(filename):**

Данная функция считывает лабиринт из текстового файла filename и возвращает его высоту h, ширину w и двумерный список maze, который представляет структуру лабиринта.

Функция начинается с открытия файла filename по адресу, заданному строкой.

Далее функция считывает все строки из файла в список lines. При этом w будет равно длине первой строки, измеренной без учета символа новой строки, так как эта строка представляет структуру лабиринта.

Высота h, то есть количество строк в лабиринте, равна длине списка lines.

Далее происходит обработка каждой строки лабиринта. С помощью генератора списка функция превращает каждый символ строки в 0, если это стена "#", или 1, если это проходимая клетка. Это необходимо для дальнейшей работы алгоритмов поиска пути.

Сгенерированный список tmp\_line добавляется в список maze, который представляет собой двумерный список, представляющий лабиринт.

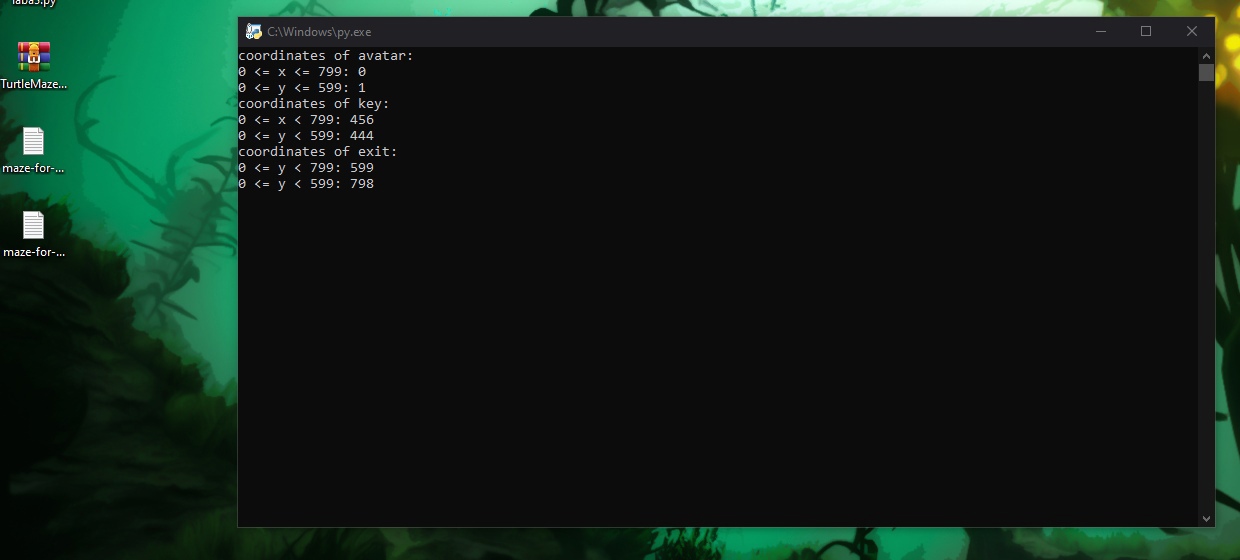
**write\_maze(filename, h, w, maze, key\_pos, path\_k, path\_e):**

Данная функция записывает лабиринт и маршруты поиска в текстовый файл filename.

Функция начинается с открытия файла filename на запись.

Далее функция проходит по строкам и столбцам лабиринта с помощью двух циклов for. Если текущая позиция (x, y) соответствует позиции ключа key\_pos, то в файл записывается символ "\*", обозначающий позицию ключа. Если текущая позиция (x, y) встречается в списке path\_k, то в файл записывается символ ".", обозначающий путь к ключу. Если текущая позиция (x, y) встречается в списке path\_e, то в файл записывается символ ",", обозначающий путь к выходу. Если текущая позиция (x, y) равна 0, то в файл записывается символ "#", обозначающий стену. Если текущая позиция является проходимой, то в файл записывается пробел.

# Пример работы алгоритма



*Рисунок 1. Ввод координат*

**Заключение**

В рамках выполнения данной работы были изучены теоретические основы структурного программирования, жадного алгоритма и алгоритма А\*.

Написана программа в структурной парадигме на языке Python, реализующая данные алгоритмы для поиска маршрута в лабиринте. Результаты обходов лабиринта и получившиеся маршруты были сохранены в файл.

Цель курсовой работы достигнута. Результатом – это программа, способная находить маршрут в лабиринте с помощью алгоритмов обхода графа с сохранением полученных маршрутов в файле. Работа программы была проверена на нескольких тестовых лабиринтах и показала хорошие результаты.

# Список литературы

1. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е. — М.: Вильямс, 2005. — 1296 с.
2. Авачева Т. Г., Пруцков А. В. Современный взгляд на концепцию структурного программирования // Cloud of Science. — 2019. — Т. 6. — № 4. — С. 646–665.
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 - URL: <https://cyberleninka.ru/article/n/algoritmy-resheniya-zadachi-bystrogo-poiska-puti-na-geograficheskih-kartah/viewer>
4. Жадные алгоритмы// Habr/[Электронный ресурс]. URL: <https://habr.com/en/articles/120343/> (Дата обращения: 30.05.2023)
5. Макконнелл С. Совершенный код. Мастер класс / Пер. с англ - М. : Издательство «Русская редакция», 2010. — 896 стр.
6. Структурное , процедурное программирование, Переход к ООП и сравнение// Центр исследования искусственного интеллекта "ЕЦИИИ"/ [Электронный ресурс]. URL: <https://intellect.icu/strukturnoe-protsedurnoe-programmirovanie-perekhod-k-oop-i-sravnenie-185> (Дата обращения: 30.05.2023)
7. A\* // Википедиа / [Электронный ресурс]URL: [https://ru.wikipedia.org/wiki/A\*](https://ru.wikipedia.org/wiki/A*) (Дата обращения: 31.05.2023)

# Приложения

**Приложение 1**

**Листинг программы**

def get\_neighbors(point, maze):

x, y = point

neighbors\_list = []

if x > 0:

neighbors\_list.append((x - 1, y))

if y > 0:

neighbors\_list.append((x, y - 1))

if x < len(maze) - 1:

neighbors\_list.append((x + 1, y))

if y < len(maze[0]) - 1:

neighbors\_list.append((x, y + 1))

return neighbors\_list

def manhattan\_distance(point1, point2):

return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])

def path(came\_from, current):

total\_path = [current]

while current in came\_from:

current = came\_from[current]

total\_path.insert(0, current)

return total\_path

def greedy(maze, start, end):

open\_list = [(start, 0)]

closed\_list = set()

came\_from = {}

while len(open\_list) > 0:

current, \_ = open\_list.pop(0)

if current == end:

return path(came\_from, end)

closed\_list.add(current)

for neighbor in get\_neighbors(current, maze):

if neighbor in closed\_list or maze[neighbor[0]][neighbor[1]] == 0:

continue

if neighbor not in [x[0] for x in open\_list]:

came\_from[neighbor] = current

priority = manhattan\_distance(neighbor, end)

open\_list.append((neighbor, priority))

open\_list.sort(key=lambda x: x[1])

return None

def astar(maze, start, end):

open\_list = [start]

closed\_list = []

g\_scores = {start: 0}

f\_scores = {start: manhattan\_distance(start, end)}

came\_from = {}

while open\_list:

current = min(open\_list, key=lambda x: f\_scores[x])

if current == end:

return path(came\_from, end)

open\_list.remove(current)

closed\_list.append(current)

for neighbor in get\_neighbors(current, maze):

if neighbor in closed\_list or maze[neighbor[0]][neighbor[1]] == 0:

continue

tentative\_g\_score = g\_scores[current] + 1

if neighbor not in open\_list:

open\_list.append(neighbor)

elif tentative\_g\_score >= g\_scores[neighbor]:

continue

came\_from[neighbor] = current

g\_scores[neighbor] = tentative\_g\_score

f\_scores[neighbor] = g\_scores[neighbor] + manhattan\_distance(neighbor, end)

return None

def read\_maze(filename):

with open(filename, "r") as file:

lines = file.readlines()

h = len(lines)

w = len(lines[0]) - 1

maze = []

for line in lines:

tmp\_line = [0 if sym == "#" else 1 for sym in line[:-1]]

maze.append(tmp\_line[:])

return h, w, maze

def write\_maze(filename, h, w, maze, key\_pos, path\_k, path\_e):

with open(filename, "w") as file:

for x in range(h):

for y in range(w):

if (x, y) == key\_pos:

file.write("\*")

elif (x, y) in path\_k:

file.write(".")

elif (x, y) in path\_e:

file.write(",")

elif maze[x][y] == 0:

file.write("#")

else:

file.write(" ")

file.write("\n")

input\_file\_name = "maze-for-u.txt"

output\_file\_name = "maze-for-me-done.txt"

h, w, maze = read\_maze(input\_file\_name)

print("coordinates of avatar:")

avatar\_x, avatar\_y = int(input(f"0 <= x <= {w - 1}: ")), int(input(f"0 <= y <= {h - 1}: "))

avatar\_pos = (avatar\_x, avatar\_y)

print("coordinates of key:")

key\_x, key\_y = int(input(f"0 <= x < {w - 1}: ")), int(input(f"0 <= y < {h - 1}: "))

key\_pos = (key\_x, key\_y)

print("coordinates of exit:")

exit\_x, exit\_y = int(input(f"0 <= y < {w - 1}: ")), int(input(f"0 <= y < {h - 1}: "))

exit\_pos = (exit\_x, exit\_y)

path\_key = greedy(maze, avatar\_pos, key\_pos)

path\_exit = astar(maze, key\_pos, exit\_pos)

write\_maze(output\_file\_name, h, w, maze, key\_pos, path\_key, path\_exit)

"""

0

1

456

444

599

798

"""