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»

Выполнила студентка 1 курса гр. ИС-33 Лядова А.Р.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc1)

[Задачи 3](#_Toc2)

[1.Теоретическая часть 4](#_Toc3)

[2. Реализация алгоритма 8](#_Toc4)

[Пример работы 11](#_Toc5)

[Заключение 12](#_Toc6)

[Список литературы 1](#_Toc7)3

[Приложение 1 1](#_Toc8)4

[Листинг программы 1](#_Toc9)4

# Введение

В данной работе мы рассмотрим два важных элемента программирования: алгоритмы поиска пути и структурное программирование. Алгоритмы поиска пути играют важную роль в разных областях, где решения должны быть максимально оптимальными - это касается логистики, игр и др. Структурное программирование же перспективно для того, чтобы добиться улучшения читаемости, тестируемости и обслуживания кода программы. Работа будет включать в себя алгоритмы жадного поиска и алгоритм А\*. Мы будем осуществлять их с помощью следующих принципов структурного программирования: использование функций и разделение программы на модули. В итоге, мы создадим программу на языке Python, которая реализует данные алгоритмы поиска пути в структурной парадигме программирования.

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

# Задачи

1. Изучить, что такое структурное программирование;
2. Изучить теорию алгоритмов обхода графа: 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.
5. Разобрать работу написанного кода, объяснить принцип работы алгоритмов и структуру программы.

# 1.Теоретическая часть

**Жадный алгоритм**

**Жадными** называют класс алгоритмов, заключающихся в принятии локально оптимальных решений на каждом этапе. Так как локально оптимальное решение вычислить гораздо проще, чем глобально оптимальное, такие алгоритмы обычно имеют хорошую асимптотику [2].

Основная идея жадного алгоритма заключается в том, что на каждом шаге алгоритм выбирает наилучшее решение по локальным условиям, без обратной связи по уже пройденному пути. То есть, из предложенных вариантов выбирается тот, который даёт лучший результат в данный момент, без учёта будущих последствий.

Жадный алгоритм позволяет получить оптимальное решение задачи путем осуществления ряда выборов. В каждой точке принятия решения в алгоритме делается выбор, который в данный момент выглядит самым лучшим. Эта эвристическая стратегия не всегда дает оптимальное решение[3-453].

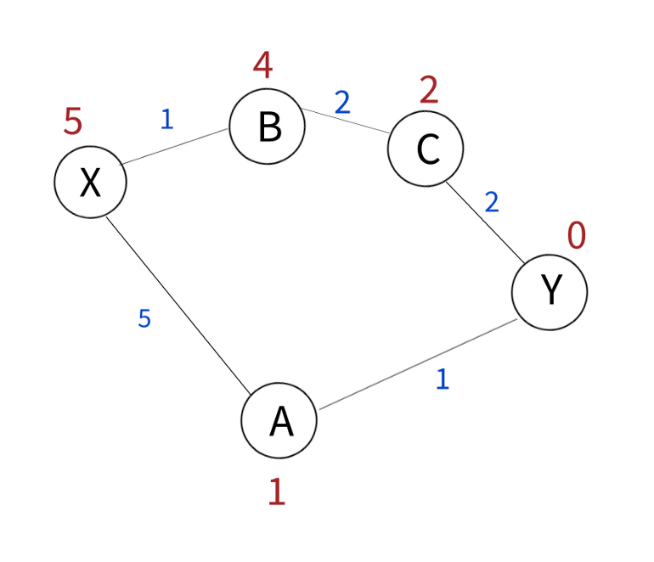
Тем не менее, в некоторых задачах жадный алгоритм может давать хороший результат, особенно когда задача имеет определенную структуру и легко поддаётся локальной оптимизации.

**Алгоритм А\***

**Алгоритм A\*** (A-star) является эффективным алгоритмом поиска кратчайшего пути в графе с использованием эвристической функции для оценки стоимости перехода от текущей вершины к целевой вершине**.**

A\* похож на алгоритм Дейкстры в том смысле, что его можно использовать для поиска кратчайшего пути. A\* похож на жадный поиск наилучшего первого поиска в том, что он может использовать эвристику для управления собой.

Секрет его успеха заключается в том, что он сочетает в себе части информации, которые использует алгоритм Дейкстры (предпочтение вершин, близких к начальной точке), и информацию, которую использует жадный поиск наилучшего первого поиска (предпочтение вершин, близких к цели). В стандартной терминологии, используемой при разговоре об A\*, g(n) представляет собой *точную стоимость* пути от начальной точки до любой вершины n и h(n) представляет собой эвристическую *оценку стоимости* от вершины n до цели. На приведенных выше диаграммах желтый (h) представляет вершины, удаленные от цели, а бирюзовый (g) представляет вершины, далекие от начальной точки. A\* уравновешивает их по мере продвижения от начальной точки к цели. Каждый раз в основном цикле проверяется вершина n с наименьшим значением f(n) = g(n) + h(n) [6].

Разберем пример. Дан граф (рис.1):

*Рисунок 1. Граф*

Допустим, мы хотим попасть из точки X в точку Y. Так как вершина Х не меняет своего положения, мы можем отбросить g(n) — ее значение равно 0. Эвристическое значение этой вершины — 5.

В подобных задачах эвристическое значение —  стоимость достижения рассматриваемой вершины из начальной.

Из вершины Х есть два пути.

Если мы перейдем в вершину А, g(n) будет равна 5 (стоимость пути), так как мы перемещаемся в новую вершину. Значение h(n) теперь равно 1. Значение f(n) в точке А будет равно 5+1 = 6. Теперь найдем значение f(n) каждой точки:

* X— A => g(A) + f(A) = 5 + 1 = 6,
* A — Y=> g(Y) + f(Y) = 6+ 0= 6,
* X— B => g(B) + f(B) = 1+ 4= 5,
* B — C => g(C) + f(C) = 3+ 2= 5,
* C — Y=> g(Y) + f(Y) = 5 + 0= 5,

Как видно из наших вычислений, кратчайший путь — X-B-C-Y. Его стоимость равна 5, в то время, как X-A-Y — 6 [1].

**Структурное программирование**

Структурное программирование является одним из фундаментальных подходов к разработке программного обеспечения. Оно предоставляет методологические инструменты и принципы для построения программных систем с понятной и легко поддерживаемой структурой. В данном докладе мы рассмотрим основные принципы структурного программирования и его влияние на развитие современных языков программирования.

Основные принципы структурного программирования:

* Использование последовательных блоков кода;
* Применение условных операторов и циклов;
* Избегание использования операторов безусловного перехода;
* Разделение программы на функции или подпрограммы;
* Принцип единственного входа и выхода.

Преимущества структурного программирования:

* Упрощение понимания и отладки кода;
* Облегчение поддержки и модификации программы;
* Повышение читаемости и структурированности кода;
* Снижение вероятности ошибок при разработке программного обеспечения [4].

# 2. Реализация алгоритма

Сначала создаем функцию **read\_maze**, которая на вход получает текстовый файл с лабиринтом, а возвращает его в виде двумерного массива **maze**.

Создаем ключ в рандомном месте массива **maze**. Далее, создаем функцию **get****\_neighbors**, которая на вход получает сам лабиринт **maze** и возвращает список **valid\_neighbors**, состоящий из кортежей, в которых хранятся координаты соседних ячеек, в которые можно перейти.

Создаем функцию **get\_heuristic**, которая вычисляет эвристическое расстояние от текущей ячейки до конечной точки в алгоритме.

Входными параметрами функции являются координаты текущей ячейки **cell** и координаты конечной точки **end**.

Функция вычисляет расстояние между текущей ячейкой и конечной точкой по формуле 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.

**Жадный алгоритм**

Создаем функцию **find\_path\_greedy**.

Входным параметром функции является сам лабиринт **maze**.

В начале задаются точки старта и конца поиска в переменных **start** и **key** соответственно.

Затем создаётся очередь **stack**, в которую добавляется кортеж **(start, [start])**. В этом кортеже первый элемент — координаты начальной точки, а второй — путь до неё.

Далее создаётся множество **visited**, в которое будут добавляться посещённые точки.

Запускается цикл, который продолжается до тех пор, пока очередь **stack** не станет пустой.

На каждой итерации цикла извлекается из очереди первый элемент **current** и путь **path** до него.

Если **current** равен точке **key**, значит путь найден и он возвращается.

Иначе точка **current** добавляется в **visited**, и для каждого соседа **neighbor** точки **current** вызывается функция **get\_neighbors**.

Сортируется список соседних позиций(**neighbors)** по возрастанию их эвристического расстояния с помощью функции **get\_heuristic** до ключа.

Для каждого из этих соседей проверяется, что он ещё не посещён. Если это так, то в очередь **stack** добавляется кортеж **(neighbor, path + [neighbor])**, где **neighbor** — это координаты соседней точки, а **path + [neighbor]** — это путь до неё.

Поиск заканчивается, если весь лабиринт пройден, но если не был найден ключ, в таком случае функция возвращает **None**.

**Алгоритм А\***

Создаем функцию **find\_path\_a\_star**.

Входным параметром функции является сам лабиринт **maze**.

Сначала создаются начальная и конечная точки **key** и **end**. Затем создается приоритетная очередь **queue**, в которую добавляется кортеж, содержащий стоимость пути, текущую точку и путь к текущей точке.

Далее создается множество **visited**, в котором будут храниться посещенные точки, чтобы избежать повторных посещений. В цикле **while**, пока очередь не пуста, извлекается кортеж с минимальной стоимостью из приоритетной очереди.

Затем проверяется, является ли текущая точка конечной точкой. Если да, то функция возвращает стоимость пути и путь к конечной точке.

Если текущая точка не является конечной точкой, то она добавляется в множество **visited**, чтобы не посещать ее повторно. Далее для каждого соседа текущей точки вызывается функция **get\_neighbors**.

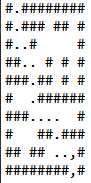
Если соседняя точка не была посещена ранее, создается новый путь к этой точке, добавляя ее в путь к текущей точке. Затем вычисляется стоимость нового пути, используя формулу **priority = len(new\_path) + get\_heuristic(neighbor, end)**. Здесь **len(new\_path)** - это фактическая стоимость пути от начальной точки до текущей, а **get\_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.

Наконец, создается новый кортеж с общей стоимостью, соседней точкой и новым путем, и добавляется в приоритетную очередь. Это продолжается до тех пор, пока конечная точка не будет достигнута, или пока приоритетная очередь не опустеет.

Если конечная точка не может быть достигнута из точки **key**, функция возвращает **None**.

В итоге функция **main** вызывают все вышеперечисленные функции. Жадный поиск выполняется от начальной точки до ключевой точки, A\* выполняется от ключевой точки до конечной точки. Затем путь, найденный жадным поиском, помечается символом ".", а путь, найденный A\*, помечается символом ",".

# Пример работы



Здесь мы наблюдаем, что из начала лабиринта идут точки, до места, где находится ключ, а после нахождения ключа уже идут запятые, до выхода из лабиринта.

# Заключение

В процессе выполнения данной работы была написана программа в структурной парадигме на языке Python, реализующая алгоритмы для поиска маршрута в лабиринте. Для написания программы были изучены алгоритмы, такие как: А\*, жадный.

В результате была разработана программа, которая на выходе создает текстовый файл, который содержит путь от начальной координаты до конечной (нарисованный точками “.” – жадный алгоритм; нарисованный запятыми “,” – А\* алгоритм).

# Список литературы

1. Алгоритм A\* и его реализация на Python // Pythonist/ [Электронный ресурс]. URL: <https://pythonist.ru/algoritm-a-star-i-ego-realizacziya-na-python/> (Дата обращения: 30.05.2023)
2. Жадные алгоритмы // Алгоритмика/ [Электронный ресурс]. URL: <https://ru.algorithmica.org/cs/combinatorial-optimization/greedy/> (Дата обращения: 31.05.2023).
3. Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. В24 Алгоритмы: построение и анализ, 2-е издание.: Пер. с англ. — М.: Издательский дом “Вильямс”, 2011. — 1296 с.: ил. — Парал. тит. Англ (Дата обращения 29.05.2023).
4. Структурное программирование / [Электронный ресурс.]. URL: <http://www.pmbk.ru/lister/035/99/index.shtml> (Дата обращения: 31.05.2023).
5. Amit Patel - "Amit's A\* Pages" / [Электронный ресурс]. URL: <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#the-a-star-algorithm> (Дата обращения: 30. 05.2023)

# Приложение 1

## Листинг программы

from queue import PriorityQueue

from math import sqrt

import random

# создание списка на основе "maze-for-u.txt" файла с лабиринтом

def read\_maze(filename):

with open(filename) as f:

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

return maze

# создание ключа в рандомном месте

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

height = len(maze)

width = len(maze[0])

passages = []

for i in range(height):

for j in range(width):

if maze[i][j] == " ":

passages.append((i, j))

random\_key = random.choice(passages)

# функция, возвращающая список валидных соседей

def get\_neighbors(maze, cell):

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

# эвристическое расстояние от ячейки до ключа или конечной точки

def get\_heuristic(cell, end):

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

# Жадный алгоритм

def find\_path\_greedy(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

# алгоритм А\*

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

# создание файла "resuult.txt"

def main():

filename = "maze-for-u.txt"

maze = read\_maze(filename)

path1 = find\_path\_greedy(maze)

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

path2 = path2[1]

for place in path1:

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

res1 = ""

for line in maze:

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

for place in path2:

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

res2 = ""

for line in maze:

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

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

f.write(res2)

main()