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 Коротаев А.П.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc1)

[Задачи 3](#_Toc2)

[1.Теоретическая часть 4](#_Toc3)

[2. Реализация алгоритма 7](#_Toc4)

[Пример работы 10](#_Toc5)

[Заключение 11](#_Toc6)

[Список литературы 12](#_Toc7)

[Приложение 1 12](#_Toc8)

[Листинг программы 13](#_Toc9)

# Введение

В данной работе будут рассмотрены элементы программирования, которые имеют важное значение: алгоритмы поиска пути и структурное программирование. Алгоритмы поиска пути существенны в областях, где необходимо находить оптимальные решения, например, в логистике, играх и т.д. Структурное программирование предназначено для улучшения читаемости, тестируемости и обслуживания кода программы. Эта работа охватит такие алгоритмы поиска пути, как жадный алгоритм и алгоритм A\*. При выполнении работы будут использоваться принципы структурного программирования, такие как использование функций и разделение программы на модули. В конечном итоге, будет разработана программа, реализующая данные алгоритмы поиска пути в структурной парадигме программирования на языке Python.

**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.[4]

Основная идея жадного алгоритма заключается в том, что на каждом шаге выбирается наилучшее решение согласно определенному критерию. Это может быть, например, выбор наименьшей стоимости, наибольшей выгоды или наименьшего времени.

Жадный алгоритм обладает некоторыми преимуществами, такими как простота реализации и быстрота работы. Он может быть полезен в случаях, когда необходимо быстро получить приемлемое решение, даже если оно не является оптимальным.

Однако жадный алгоритм не всегда гарантирует достижение оптимального результата. Иногда он может привести к локально оптимальному, но не глобально оптимальному решению. Это происходит из-за того, что жадный алгоритм не учитывает возможные последствия своих решений на будущих шагах.

Чтобы использовать жадный алгоритм эффективно, важно правильно определить критерий выбора на каждом шаге и убедиться, что он приводит к приемлемому результату для данной задачи. В некоторых случаях жадный алгоритм может использоваться как первоначальное приближение для более сложных оптимизационных методов.

**Алгоритм А\***

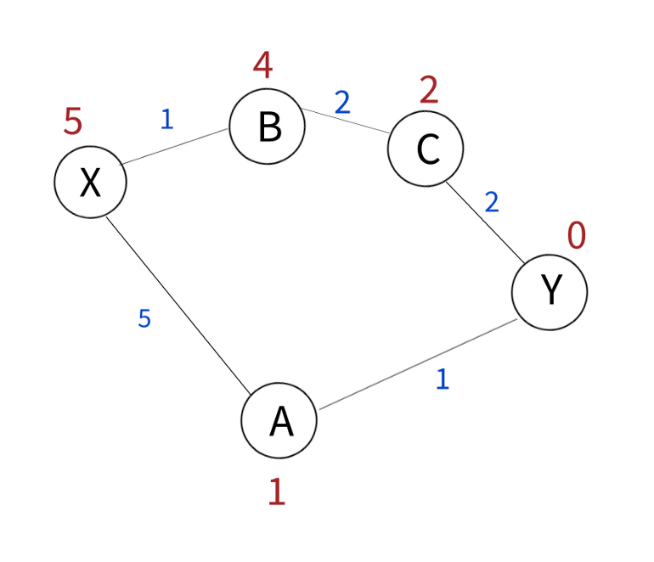
В 1964 году Нильс Нильсон изобрел эвристический подход к увеличению скорости алгоритма Дейкстры. Этот алгоритм был назван А1. В 1967 году Бертрам Рафаэль сделал значительные улучшения по этому алгоритму, но ему не удалось достичь оптимальности. Он назвал этот алгоритм A2. Тогда в 1968 году Петр Э. Харт представил аргументы, которые доказывали, что A2 был оптимальным при использовании последовательной эвристики лишь с незначительными изменениями. В его доказательство алгоритма также включен раздел, который показывал, что новый алгоритм A2 был, возможно, лучшим алгоритмом, учитывая условия.

## Принцип работы A\*

Практически, алгоритм A\* отличается от алгоритма Дейкстры направленностью обхода узлов графа за счёт использования эвристической функции, определяющей ориентировочное расстояние между данным узлом и концом пути. Иными словами, приоритет отдаётся тем узлам, которые согласно эвристической функции находятся ближе к концу пути.

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Сначала рассматриваются те маршруты, которые «кажутся» ведущими к цели. В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение эвристической функции, после чего этот узел раскрывается.

В случае с графом, 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 множественных решений выбирается решение с наименьшей стоимостью. [1]

Разберем пример. У нас есть 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.[2]

# 2. Реализация алгоритма

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

Создаем ключ в рандом 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**.

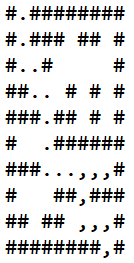
Если соседняя точка не была посещена ранее, создается новый путь к этой точке, добавляя ее в путь к текущей точке. Затем вычисляется стоимость нового пути, используя формулу **priority = len(new\_path) + get\_heuristic(neighbor, end)**. Здесь **len(new\_path)** - это фактическая стоимость пути от начальной 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 точку **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 можем увидеть, что из начала лабиринта идут точки, до места, где находится ключ, а после нахождения ключа уже идут запятые, до выхода из лабиринта.

# Заключение

В рамках выполнения данной работы была написана программа в структурной парадигме на языке Python, реализующая алгоритмы для поиска маршрута в лабиринте. Для написания программы потребовалось изучения алгоритмов, таких 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\* // studfile.net / [Электронный ресурс] URL: <https://studfile.net/preview/3239584/page:4/> (Дата обращения: 31.05.2023).
2. Алгоритм A\* и его реализация на Python // Pythonist/ [Электронный ресурс]. URL: <https://pythonist.ru/algoritm-a-star-i-ego-realizacziya-na-python/> (Дата обращения: 31.05.2023).
3. Рафгарден Тим. Совершенный алгоритм. Жадные алгоритмы и динамическое программирование. — СПб.: Питер, 2020. — 256 с. — Текст: непосредственный. (Дата обращения: 31.05.2023).
4. Жадные алгоритмы // Алгоритмика/ [Электронный ресурс]. URL: <https://ru.algorithmica.org/cs/combinatorial-optimization/greedy/> (Дата обращения: 31.05.2023).

# Приложение 1

## Листинг программы

from queue import PriorityQueue  
from math import sqrt  
import random  
  
# создаём список 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  
  
  
# создает ключ в рандом 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, которая возвращает 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)  
 # Sort neighbors based on their heuristic distance to the key  
 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  
  
# 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 res.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("res.txt", "w") as f:  
 f.write(res2)  
  
main()