МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Жадный алгоритм и алгоритм A*.

Студентка гр. 1304 _____ Ярусова Т. В.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург

2023

Цель работы.

Разработать программу, которая решает задачу построения кратчайшего пути в ориентированном графе. Изучить жадный алгоритм и алгоритм A*.

Задание.

1. Разработайте программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

2. Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A*. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

Входные данные

a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0

В первой строке через пробел указываются начальная и конечная вершины Далее в каждой строке указываются ребра графа и их вес.

Выходные данные:

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Выполнение работы.

Жадный алгоритм.

Класс Graph.

- **Метод __init__().**

Конструктор, в котором инициализируются переменные:

- *start_node* – вершина, от которой необходимо найти путь;
- *end_node* – вершина, до которой необходимо построить путь;
- *nodes* – словарь, в котором ключи - вершины, а значения - примыкающие вершины и стоимость пути.
- *path* – строка, в которой будет храниться путь от начальной вершины start_node до конечной вершины end_node.

- **Метод read().**

В данном методе происходит считывание входных данных и заполнение полей *start_node, end_node, nodes*. Считывание происходит в цикле *while* с помощью конструкции *try-except,* которая позволяет закончить считывание при обнаружении пустой строки.

- **Метод greedy_algorithm().**

Метод, реализующий поиск кратчайшего пути в ориентированном графе жадным алгоритмом.

Для каждой вершины происходит сортировка примыкающих к ней вершин по весу пути. После этого в цикле *while* до тех пор пока текущей вершиной не станет конечная вершина производится оптимальный выбор вершины, который заключается во взятии первой примыкающей вершины, так как они были отсортированы по стоимости пути.

- **Метод print_answer().**

В данном методе происходит вывод переменно *path,* заполненной после вызова метода *greedy_algorithm().*

Функция solve().

В данной функции создается экземпляр класса *Graph*. Вызывается метод *read()* для считывания входных данных и вызывается метод

greedy_algorithm() для поиска кратчайшего пути. А также вызывается метод *print_answer(),* который печатает ответ на экран.

Алгоритм A*.

Класс Graph.

- **Метод __init__().**

Конструктор, в котором инициализируются переменные:

- *start_node* – вершина, от которой необходимо найти путь;
- *end_node* – вершина, до которой необходимо построить путь;
- *nodes* – словарь, в котором ключи - вершины, а значения - примыкающие вершины и стоимость пути.

path – строка, в которой будет храниться путь от начальной вершины start_node до конечной вершины end_node.

- **Метод read().**

В данном методе происходит считывание входных данных и заполнение полей *start_node, end_node, nodes*. Считывание происходит в цикле *while* с помощью конструкции *try-except,* которая позволяет закончить считывание при обнаружении пустой строки.

- **Метод heuristics().**

Метод, реализующий эвристическую функцию. Возвращает модуль разницы между кодами символов следующей вершины и конечной вершины.

- **Метод get_neighbors().**

Данный метод возвращает список кортежей примыкающих вершин к текущей.

- **Метод build_path().**

Метод, в котором происходит построение пути по данным, хранящимся в словаре *came_from*. Ключом является вершина, в которую пришли, а значением – из которой пришли.

После заполнения переменой *path,* данная строка «переворачивается» и выводится на экран.

- **Метод a_star().**

Данный метод реализует поиск кратчайшего пути в ориентированном графе алгоритмом A*.

В данном методе объявляются вспомогательные переменные *nodes_queue* – очередь с приоритетом, элементами которой будут название вершин и стоимость пути в данную вершину от начальной + значение эвристической функции, *came_from* – словарь, в котором ключ – вершина, в которую пришли, а значение – вершина, из которой пришли, *cost_so_far* – словарь, в котором ключ название вершины, а значение – стоимость пути от начальной вершины до текущей.

Основной алгоритм реализуется в цикле *while*. Цикл работает до тех пор пока очередь *nodes_queue* не пустая, либо он прерывается, если текущая выбранная вершина является финальной вершиной. Текущей вершиной становится первая вершина, которая хранится в очереди *nodes_queue*. С помощью цикла *for* совершается проход по всем прилегающим вершинам к текущей и пересчитывается значение стоимости пути для каждой прилегающей вершины. В очередь кладется вершина и ее стоимость пути.

После окончания цикла *while* вызывается метод *build_path(),* который построит путь по данным словаря *came_from* и выведет результат на экран.

Функция solve().

В данной функции создается экземпляр класса *Graph*. Вызывается метод *read()* для считывания входных данных и вызывается метод *greedy_algorithm()* для поиска кратчайшего пути.

Разработанный программный код см. в приложении А.

Выводы.

В ходе лабораторной работы изучены жадный алгоритм и алгоритм A* для нахождения пути в графе. На языке программирования Python реализованы данные алгоритмы поиска кратчайшего пути.

Для оптимизации работы жадного алгоритма словарь, в котором хранились ребра ориентированного графа, был отсортирован по весам, что позволило выбирать на каждом шаге оптимальный вариант.

Для реализации алгоритма A* использовался модуль queue для создания очереди с приоритетом, которая позволила выбирать оптимальные вершины в момент поиска пути.

Также в процессе работы над алгоритмами отмечено, что жадный алгоритм в больших случаях не подходит для нахождения глобального оптимального решения в данном виде задач в отличие от алгоритма A*

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Алгоритм A* // habr.com URL: https://habr.com/ru/post/331192/

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: greedy_algorithm.py

```
class Graph:

    # Инициализация класса
    def __init__(self):
        self.start_node = None
        self.end_node = None
        self.nodes = dict()
        self.path = ''

    # Метод, в котором происходит считывание входных данных
    def read(self):
        self.start_node, self.end_node = input().split()
        while True:
            try:
                from_node, in_node, weight = input().split()
                if from_node in self.nodes:
                            self.nodes[from_node].append([in_node,
float(weight)])
                else:
                              self.nodes[from_node]  =  [[in_node,
float(weight)]]
            except:
                break

    # Метод, реализующий поиск кратчайшего пути жадным алгоритмом
    def greedy_algorithm(self):
        for key in self.nodes.keys():
            self.nodes[key].sort(key=lambda elem: elem[1])

        self.path = self.start_node
        while self.path[-1] != self.end_node:
            current_node = self.path[-1]
                    if  (self.nodes[current_node][0][0]  not  in
self.nodes.keys()) and \
                                (self.nodes[current_node][0][0]  !=
self.end_node):
                self.nodes[current_node].pop(0)
            self.path += self.nodes[current_node][0][0]

    # Функция, которая выводит путь на экран
    def print_answer(self):
        print(self.path)

# Функция, в которой создается экземпляр класса Graph
# и вызываются методы, которые реализуют поиск кратчайшего пути
жадным алгоритмом
def solve():
    graph = Graph()
    graph.read()
    graph.greedy_algorithm()
    graph.print_answer()
```

```
solve()
```

Название файла: a_star.py

```
import queue

class Graph:
    # Инициализация класса
    def __init__(self):
        self.start_node = None
        self.end_node = None
        self.nodes = dict()
        self.path = ''

    # Метод, в котором происходит считывание входных данных
    def read(self):
        self.start_node, self.end_node = input().split()
        while True:
            try:
                from_node, in_node, weight = input().split()
                if from_node in self.nodes:
                            self.nodes[from_node].append([in_node,
float(weight)])
                else:
                                self.nodes[from_node]  =  [[in_node,
float(weight)]]
            except:
                break

    # Метод, реализующий эвристическую функцию
      # На вход принимает следующую вершины, в которую можно
перейти
     # Возвращает разницу между кодами символов финальной вершины
и вершины, поступившей на вход
    def heuristics(self, node):
        return abs(ord(node) - ord(self.end_node))

      # Метод, который возвращает список кортежей примыкающих
вершин к текущей
    def get_neighbors(self, node):
        return self.nodes[node]

    # Метод, который строит и выводит путь, найденный алгоритмом
A*
     # На вход получает словарь, составленный по принципу откуда
пришли к текущей вершине
    def build_path(self, came_from):
        current_node = self.end_node
        while current_node != None:
            self.path += current_node
            current_node = came_from[current_node]
        self.path = self.path[::-1]
        print(self.path)

    # Метод, реализующий алгорит A*
    def a_star(self):
```

```
            nodes_queue = queue.PriorityQueue()
            nodes_queue.put((0, self.start_node))

            came_from = {}
            came_from[self.start_node] = None

            cost_so_far = {}
            cost_so_far[self.start_node] = 0

            while not nodes_queue.empty():
                current_node = nodes_queue.get()[1]

                if current_node == self.end_node:
                    break

                if current_node in self.nodes.keys():
                    neighbors = self.get_neighbors(current_node)
                    for next_node, cost_next_node in neighbors:
                                new_cost = cost_so_far[current_node] +
cost_next_node
                        if next_node not in cost_so_far or new_cost <
cost_so_far[next_node]:
                            cost_so_far[next_node] = new_cost
                                        priority = new_cost +
self.heuristics(next_node)
                            nodes_queue.put((priority, next_node))
                            came_from[next_node] = current_node

            self.build_path(came_from)

    # Функция, в которой создается экземпляр класса Graph
    # и вызываются методы, которые реализуют поиск кратчайшего пути
алгоритмом A*
    def solve():
        graph = Graph()
        graph.read()
        graph.a_star()

    solve()
```