

**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 A\***  
**Вариант 9**

Студентка гр. 1381

\_\_\_\_\_

Рымарь М.И.

Преподаватель

\_\_\_\_\_

Токарев А.П.

Санкт-Петербург

2023

### **Цель работы.**

Изучить алгоритм построения пути в ориентированном графе. Написать программы, первая из которых решает задачу построения пути в орграфе при помощи жадного алгоритма, вторая находит кратчайший путь в орграфе методом A\*.

### **Задание.**

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.

Пример входных данных

*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.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

*abcde*

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.

Пример входных данных

*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.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

*ade*

Индивидуализация (9): вывод графического представления графа.

### **Основные теоретические положения.**

Жадные алгоритмы – класс алгоритмов, в которых на каждом шаге делается локально оптимальный выбор в надежде, что это приведёт к оптимальному решению.

Алгоритм  $A^*$  — это модификация алгоритма Дейкстры, оптимизированная для единственной конечной точки. Алгоритм Дейкстры может находить пути ко всем точкам,  $A^*$  находит путь к одной точке. Он отдаёт приоритет путям, которые ведут ближе к цели.

### **Выполнение работы.**

1. Жадный алгоритм поиска кратчайшего пути в ориентированном графе.

Сначала пользователю предлагается ввести начальную и конечную вершины графа. Затем программа считывает входные данные, представляющие собой описание ребер графа и их весов. Ребра хранятся в списке `list_nodes`, каждый элемент которого представляет собой тройку значений: начальную вершину, конечную вершину и вес ребра.

Функция `greedy()` принимает на вход список `list_nodes`, а также начальную и конечную вершины. Она использует жадный алгоритм поиска кратчайшего пути, который на каждом шаге выбирает ближайшую вершину и добавляет ее к кратчайшему пути. Алгоритм продолжается, пока не будет найден путь от начальной до конечной вершины.

На каждой итерации цикла `while` программа перебирает все ребра графа, которые начинаются в текущей вершине `cur` и выбирает ребро с минимальным весом, которое ведет к непосещенной вершине. Если такое ребро находится, то программа перемещается к этой вершине и добавляет ее в кратчайший путь. Если же такого ребра нет, программа возвращает последнюю добавленную вершину из кратчайшего пути и продолжает поиск пути из этой вершины.

По завершении работы алгоритма программа возвращает строку, содержащую кратчайший путь от начальной до конечной вершины.

2. Алгоритм  $A^*$  для поиска кратчайшего пути в графе между двумя заданными вершинами.

1). Сначала задаются функция эвристической оценки расстояния между двумя вершинами и вспомогательные структуры данных.

2). Затем в основном цикле `while` извлекается вершина из очереди с приоритетом с минимальной оценкой, и, если эта вершина является искомой конечной вершиной, цикл прерывается и происходит формирование ответа.

3). В противном случае перебираются все ребра, исходящие из текущей вершины. Для каждого ребра вычисляется новое расстояние до соседней вершины (в зависимости от веса ребра) и если это расстояние меньше, чем расстояние до этой соседней вершины, вычисляется новая оценка приоритета для соседней вершины и она добавляется в очередь с приоритетом.

4). В конце вычисляется и возвращается путь от начальной до конечной вершины, обратив список предшествующих вершин.

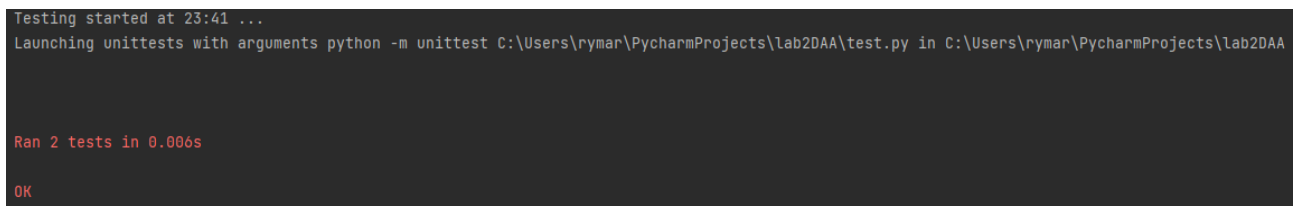
5). Входные данные представлены в виде ввода пользователя, где первая строка содержит начальную и конечную вершины, а затем последовательно вводятся строки с описанием ребер и их весов в формате "узел1 узел2 вес". Выводится найденный кратчайший путь в формате строки.

Исходный код программ представлен в приложении А.

### **Тестирование.**

Программа, протестированная с помощью unittest, гарантирует, что ее функции работают должным образом и выполняют все задачи, для которых они были написаны. При тестировании были рассмотрены различные случаи, включая граничные, чтобы убедиться, что программа работает корректно во всех возможных сценариях использования.

Результаты выполнения тестов представлены на рисунке 1. Программный код приведён в приложении Б.



```
Testing started at 23:41 ...  
Launching unittests with arguments python -m unittest C:\Users\rymar\PycharmProjects\lab2DAA\test.py in C:\Users\rymar\PycharmProjects\lab2DAA  
  
Ran 2 tests in 0.006s  
  
OK
```

Рисунок 1 – Результаты выполнения тестов

### **Исследование алгоритма.**

Теоретическая сложность жадного алгоритма поиска пути в графе зависит от конкретной задачи и структуры графа. В худшем случае жадный алгоритм может потребовать экспоненциального времени для нахождения оптимального пути. Например, если граф имеет множество циклов и есть несколько возможных путей между двумя вершинами, то жадный алгоритм может заиклиться и не сможет найти оптимальный путь.

Однако, в большинстве случаев жадный алгоритм является эффективным и быстрым. Практическая сложность жадного алгоритма зависит от размера графа и его структуры. В некоторых случаях, жадный алгоритм может давать хорошие результаты, но в других случаях, например, когда граф имеет много локальных оптимумов, жадный алгоритм может давать неверные результаты.

Таким образом, практическая сложность жадного алгоритма поиска пути в графе зависит от конкретной задачи, структуры графа и того, какие дополнительные условия наложены на поиск пути.

Теоретическая сложность алгоритма  $A^*$  зависит от размера графа и сложности эвристики. В худшем случае, если эвристика не является информативной и не может существенно сократить пространство поиска, алгоритм  $A^*$  может иметь экспоненциальную сложность. Однако, при использовании информативной эвристики, алгоритм  $A^*$  имеет полиномиальную сложность.

Практическая сложность алгоритма  $A^*$  зависит от размера графа, его структуры, эвристики и реализации алгоритма. Некоторые варианты алгоритма  $A^*$  могут иметь лучшую производительность в конкретных условиях. Кроме того, выбор оптимальной эвристики может оказать значительное влияние на производительность алгоритма  $A^*$ .

### **Визуализация.**

Визуализация графа была произведена с помощью graphviz. Graphviz - это библиотека для визуализации графов и ориентированных графов (орграфов). Она может использоваться в Python с помощью модуля PyGraphviz.

Чтобы визуализировать орграф с помощью Graphviz в Python, необходимо определить вершины и дуги графа, а затем использовать функцию "render" из PyGraphviz для создания изображения. Это позволяет создавать красивые и интуитивно понятные графические представления сложных орграфов.

Пример визуализации графа представлен на рисунке 2. Программный код приведён в приложении В.

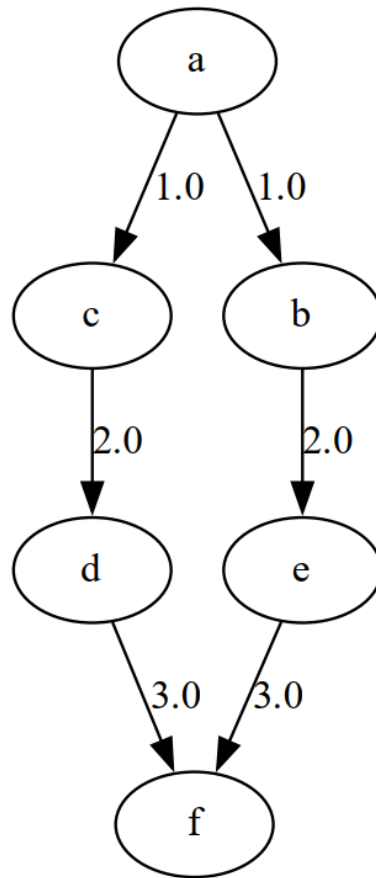


Рисунок 2 – Визуализация орграфа

### **Выводы.**

В ходе выполнения лабораторной работы изучен алгоритм поиска кратчайшего пути жадным алгоритмом и алгоритмом А\*. Написана программа, реализующая эти алгоритмы. Исследована их теоретическая и практическая сложность. Выполнена визуализация графа.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ ПРОГРАММНЫЙ КОД

Название файла: greedyAlg.py

```
def greedy(graph, start, end):
    cur, res = start, [start]
    visited = [False] * 30
    visited[ord(cur) - ord('a')] = True
    while cur != end:
        min = float("inf")
        flag = False
        next_node = ''
        for i in range(len(graph)):
            if cur == graph[i][0]:
                if not visited[ord(graph[i][1]) - ord('a')] and
graph[i][2] < min:
                    min = graph[i][2]
                    next_node = graph[i][1]
                    flag = True
        visited[ord(cur) - ord('a')] = True
        if not flag:
            if res:
                res.pop(-1)
                cur = res[-1]
                continue
        cur = next_node
        res.append(cur)
    return ''.join(res)

if __name__ == '__main__':
    start, end = map(str, input().split())
    list_nodes = []
    while True:
        try:
            data = input()
        except EOFError:
            break
        if not data:
            break
        node1, node2, weight = data.split()
        list_nodes.append([node1, node2, float(weight)])
    print(greedy(list_nodes, start, end))
```

Название файла: aStar.py

```
import heapq

def heuristic(a, b):
    return abs(ord(a) - ord(b))

def aStar(graph, start, end):
    sum_path, path, priority_queue = {start: 0}, {start: None}, []
    heapq.heappush(priority_queue, [0, start])

    while len(priority_queue):
```



```

        current = heapq.heappop(priority_queue)[1]
        if current == end:
            break
        for i in range(len(graph)):
            if current == graph[i][0]:
                weight = sum_path[current] + graph[i][2]
                if graph[i][1] not in sum_path or weight <
sum_path[graph[i][1]]:
                    sum_path[graph[i][1]] = weight
                    priority_number = weight + heuristic(end,
graph[i][1])
                    heapq.heappush(priority_queue, [priority_number,
graph[i][1]])
                    path[graph[i][1]] = current
        answer, previous = end, path[end]
        while previous is not None:
            answer = answer + previous
            previous = path[previous]
        return answer[::-1]

if __name__ == '__main__':
    start, end = input().split()
    graph = []
    while True:
        try:
            text = input()
        except EOFError:
            break
        if not text:
            break
        node1, node2, weight = text.split()
        graph.append([node1, node2, float(weight)])
    print(aStar(graph, start, end))

```

## ПРИЛОЖЕНИЕ Б

### ИСХОДНЫЙ КОД ТЕСТИРУЮЩЕГО ФАЙЛА

```
import unittest
import greedyAlg
import aStar

tests = [[['a', 'g'],
          ['a', 'b', 3.0], ['a', 'c', 1.0], ['b', 'd', 2.0],
          ['b', 'e', 3.0], ['d', 'e', 4.0], ['e', 'a', 1.0],
          ['e', 'f', 2.0], ['a', 'g', 8.0], ['f', 'g', 1.0]],
         [['a', 'e'],
          ['a', 'b', 7.0], ['a', 'c', 3.0], ['b', 'c', 1.0],
          ['c', 'd', 8.0], ['b', 'e', 4.0]],
         [['a', 'd'],
          ['a', 'b', 1.0], ['b', 'c', 1.0], ['c', 'a', 1.0], ['a', 'd',
8.0]],
         [['a', 'b'],
          ['a', 'b', 1.0], ['a', 'c', 1.0]],
         [['b', 'e'],
          ['a', 'b', 1.0], ['a', 'c', 2.0], ['b', 'd', 7.0],
          ['b', 'e', 8.0], ['a', 'g', 2.0], ['b', 'g', 6.0],
          ['c', 'e', 4.0], ['d', 'e', 4.0], ['g', 'e', 1.0]]]

answers1 = [['abdefg'], ['abe'], ['ad'], ['ab'], ['bge']]

answers2 = [['ag'], ['abe'], ['ad'], ['ab'], ['be']]

class MyTestCase(unittest.TestCase):
    def test_greedy(self):
        for i in range(len(answers1)):
            self.assertEqual(greedyAlg.greedy(tests[i][1:],
tests[i][0][0], tests[i][0][1]), *answers1[i])
    def test_astar(self):
        for i in range(len(answers2)):
            self.assertEqual(aStar.aStar(tests[i][1:],      tests[i][0][0],
tests[i][0][1]), *answers2[i])

if __name__ == '__main__':
    unittest.main()
```

## ПРИЛОЖЕНИЕ В

### ИСХОДНЫЙ КОД ДЛЯ ВИЗУАЛИЗАЦИИ ОРГРАФА

```
import graphviz
import time

time_pause = 0.1

def graphics(graph, start, end):
    current, next_node = start, ''
    visited = [false] * 30
    visited[ord(current) - ord('a')] = true
    scheme = graphviz.digraph()
    while current != end:
        for i in range(len(graph)):
            if current == graph[i][0] and not visited[ord(graph[i][1]) -
ord('a')]:
                scheme.node(current, fontcolor='black')
                scheme.node(graph[i][1], fontcolor='black')
                scheme.edge(current, graph[i][1], label=str(graph[i][2]))
                scheme.render(f'{time.time()}.gv', view=true)
                time.sleep(time_pause)
            visited[ord(current) - ord('a')] = true
            current = chr(ord(current) + 1)

if __name__ == '__main__':
    start, end = input().split()
    graph = []
    while True:
        try:
            text = input()
        except EOFError:
            break
        if not text:
            break
        node1, node2, weight = text.split()
        graph.append([node1, node2, float(weight)])
    graphics(graph, start, end)
```