

**МИНОБРНАУКИ РОССИИ**  
**Санкт-Петербургский государственный**  
**электротехнический университет**  
**«ЛЭТИ» им. В.И. Ульянова (Ленина)**  
**Кафедра математического обеспечения и применения ЭВМ**

**ОТЧЕТ**  
**по лабораторной работе №2**  
**по дисциплине «Построение и анализ алгоритмов»**  
**Тема: Жадный алгоритм и  $A^*$ .**

Студент гр. 8382

Терехов А.Е.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

### **Цель работы.**

Изучить алгоритм работы жадного алгоритма и алгоритма A\*.

### **Задание.**

Вариант 1. В A\* вершины именуются целыми числами (в т. ч. отрицательными).

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
```

### **Описание структур.**

С исходным кодом жадного алгоритма можно ознакомиться в приложении А.

Для хранения графа были использованы списки смежности, реализованные на базе словаря и класса Edge (листинг 1). Ключ словаря является началом дуги,

а вся остальная информация содержится в классе Edge имеющем два поля: конец дуги и стоимость передвижения. Все остальные используемые структуры такие как списки, числа, строки, символы предоставлены языком Python 3.7.

Листинг 1 – Реализация класса Edge.

```
class Edge:
    def __init__(self, e, w):
        self.end = e
        self.weight = w
    def __str__(self):
        return str(self.end) + " " + str(self.weight)
    def __lt__(self, other):
        return (self.weight < other.weight)
    def __le__(self, other):
        return (self.weight <= other.weight)
```

С исходным кодом алгоритма A\* можно ознакомиться в приложениях Б и В.

Для алгоритма A\* так же было выбрано хранение графа с использованием списков смежности только без лишних структур. Списки смежности представляют собой словарь, ключи которого это вершины, из которых существует хоть один путь. Значение – это тоже словарь, но в нем ключи – это концы дуг, а значение – стоимость. То есть это выглядит следующим образом:

```
{ начало_дуги : { конец_дуги : стоимость } }.
```

Листинг 2 – чтение графа в словарь.

```
graph = {}
for line in stdin:
    u,v,c = line.replace('\n', '').split()
    if int(u) not in graph.keys():
        graph[int(u)] = {int(v):float(c)}
    else:
        graph[int(u)][int(v)] = float(c)
```

### **Описание алгоритма.**

Жадный алгоритм очень просто устроен. В начале инициализируем путь и текущую вершину стартовой вершиной. Затем пока текущая рассматриваемая вершина не равна конечной выбираем самого дешевого соседа, добавляем его в путь, изменяем его вес на очень большой, чтобы избежать заикливания, и

переходим в соседа. Если же был встречен тупик или все соседи вершины были рассмотрены, то просто возвращаемся в предыдущую вершину и удаляем из пути тупик. Если путь станет пуст, то это скажет о том, что пути просто не существует.

$A^*$  пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые могут быть ведущими к цели. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Составляющая  $g(x)$  — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение  $f(x)$ , после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в очереди с приоритетом. Приоритет пути определяется по значению  $f(x) = g(x) + h(x)$ . Алгоритм продолжает свою работу до тех пор, пока множество раскрытых вершин не опустеет. Из множества решений выбирается решение с наименьшей стоимостью.

Множество просмотренных вершин хранится в `closedset`, а требующие рассмотрения пути — в множестве `openset`. Приоритет пути вычисляется с помощью функции  $f(x)$  путем выбора первого минимального.

### **Сложность алгоритма.**

Сложность жадного алгоритма по времени в случае, если поиск минимальной дуги происходит по списку или массиву равна  $O(|E|\log(|E|) + |V|)$ . Так как в общем случае алгоритм посещает  $|V|$  вершин и каждый поиск минимума складывается в  $|E|\log(|E|)$ .

По памяти жадный алгоритм имеет сложность  $O(|E| + |V|)$ , так как хранятся

все ребра в словаре и путь.

Для  $A^*$  все сложнее, его временная сложность зависит от используемой эвристической функции. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:  $|h(x) - h^*(x)| < O(\log(h^*(x)))$ . Где  $h^*(x)$  – оптимальная эвристика, то есть точное расстояние от  $x$  до цели. Для данного алгоритма эвристика не является даже корректной (доказано ниже в тестировании), поэтому очевидно, что данное условие не выполнено.

По памяти сложность составляет  $O(|E| + |V|)$ , так как хранятся ребра и вершины в одном из множеств `openset` или `closedset` и в словаре `fromset`.

### Тестирование.

Проверим жадный алгоритм на следующем тесте:

Таблица 1. Тестирование жадного алгоритма.

Input	Output
a l	from a to b because minimal cost of [('b', 1.0), ('f', 3.0)] is 1.0
a b 1	from b to g because minimal cost of [('g', 3.0), ('c', 5.0)] is 3.0
a f 3	from g to e because minimal cost of [('e', 4.0), ('i', 5.0)] is 4.0
b g 3	from e to h because minimal cost of [('h', 1.0), ('n', 1.0)] is 1.0
b c 5	Come back from h to e because here is a dead end.
f g 4	from e to n because minimal cost of [('h', inf), ('n', 1.0)] is 1.0
c d 6	from n to m because minimal cost of [('m', 2.0)] is 2.0
d m 1	from m to j because minimal cost of [('j', 3.0)] is 3.0
g e 4	from j to l because minimal cost of [('l', 5.0)] is 5.0
g i 5	abgenmjl
e h 1	
e n 1	
n m 2	
i k 1	
i j 6	
j l 5	
m j 3	

Для  $A^*$  были написаны два теста задающие одинаковые графы, но с разными названиями вершин, в одном это буквы, а в другом целые числа, в том числе и отрицательные.

Таблица 2. Тестирование алгоритма  $A^*$  (начало).

Input	Output
a f	a append START to openset
a b 1	openset: ['a']
a c 1	Take minimal $f(a) = 5$ from {'a': 5}
b d 2	a pop from openset
b f 6	openset: []
c b 2	a append to closedset
c e 4	closedset: ['a']
d c 3	b neighbour of a append to openset
d e 1	openset: ['b']
d f 3	To b from a
e f 4	fromset {'b': 'a'}
	c neighbour of a append to openset
	openset: ['b', 'c']
	To c from a
	fromset {'b': 'a', 'c': 'a'}
	Take minimal $f(c) = 4.0$ from {'a': 5, 'b': 5.0, 'c': 4.0}
	c pop from openset
	openset: ['b']
	c append to closedset
	closedset: ['a', 'c']
	e neighbour of c append to openset
	openset: ['b', 'e']
	To e from c
	fromset {'b': 'a', 'c': 'a', 'e': 'c'}
	Take minimal $f(b) = 5.0$ from {'a': 5, 'b': 5.0, 'c': 4.0, 'e': 6.0}
	b pop from openset
	openset: ['e']
	b append to closedset
	closedset: ['a', 'c', 'b']
	d neighbour of b append to openset
	openset: ['e', 'd']
	To d from b
	fromset {'b': 'a', 'c': 'a', 'e': 'c', 'd': 'b'}
	f neighbour of b append to openset
	openset: ['e', 'd', 'f']
	To f from b

Таблица 2. Тестирование алгоритма A\* (конец).

	fromset { 'b': 'a', 'c': 'a', 'e': 'c', 'd': 'b', 'f': 'b' } Take minimal $f(d) = 5.0$ from { 'a': 5, 'b': 5.0, 'c': 4.0, 'e': 6.0, 'd': 5.0, 'f': 7.0 } d pop from openset openset: ['e', 'f'] d append to closedset closedset: ['a', 'c', 'b', 'd'] To e from d fromset { 'b': 'a', 'c': 'a', 'e': 'd', 'd': 'b', 'f': 'b' } To f from d fromset { 'b': 'a', 'c': 'a', 'e': 'd', 'd': 'b', 'f': 'd' } Take minimal $f(e) = 5.0$ from { 'a': 5, 'b': 5.0, 'c': 4.0, 'e': 5.0, 'd': 5.0, 'f': 6.0 } e pop from openset openset: ['f'] e append to closedset closedset: ['a', 'c', 'b', 'd', 'e'] Take minimal $f(f) = 6.0$ from { 'a': 5, 'b': 5.0, 'c': 4.0, 'e': 5.0, 'd': 5.0, 'f': 6.0 } a b d f
--	--

Таблица 3. Тестирование алгоритма A\* с вершинами в целых числах (начало).

Input	Output
-1 3	-1 append START to openset
-1 1 1	openset: [-1]
-1 -2 1	Take minimal $f(-1) = 4$ from {-1: 4}
1 2 2	-1 pop from openset
1 3 6	openset: []
-2 1 2	-1 append to closedset
-2 -3 4	closedset: [-1]
2 -2 3	1 neighbour of -1 append to openset
2 -3 1	openset: [1]
2 3 3	To 1 from -1
-3 3 4	fromset { 1: -1 }
	-2 neighbour of -1 append to openset
	openset: [1, -2]
	To -2 from -1
	fromset { 1: -1, -2: -1 }
	Take minimal $f(1) = 3.0$ from {-1: 4, 1: 3.0, -2: 6.0}
	1 pop from openset
	openset: [-2]
	1 append to closedset
	closedset: [-1, 1]
	2 neighbour of 1 append to openset
	openset: [-2, 2]
	To 2 from 1

Таблица 3. Тестирование алгоритма  $A^*$  с вершинами в целых числах (конец).

<p> fromset {1: -1, -2: -1, 2: 1}  3 neighbour of 1 append to openset  openset: [-2, 2, 3]  To 3 from 1  fromset {1: -1, -2: -1, 2: 1, 3: 1}  Take minimal <math>f(2) = 4.0</math> from {-1: 4, 1: 3.0, -2: 6.0, 2: 4.0, 3: 7.0}  2 pop from openset  openset: [-2, 3]  2 append to closedset  closedset: [-1, 1, 2]  -3 neighbour of 2 append to openset  openset: [-2, 3, -3]  To -3 from 2  fromset {1: -1, -2: -1, 2: 1, 3: 1, -3: 2}  To 3 from 2  fromset {1: -1, -2: -1, 2: 1, 3: 2, -3: 2}  Take minimal <math>f(-2) = 6.0</math> from {-1: 4, 1: 3.0, -2: 6.0, 2: 4.0, 3: 6.0, -3: 10.0}  -2 pop from openset  openset: [3, -3]  -2 append to closedset  closedset: [-1, 1, 2, -2]  Take minimal <math>f(3) = 6.0</math> from {-1: 4, 1: 3.0, -2: 6.0, 2: 4.0, 3: 6.0, -3: 10.0}  -1 1 2 3 </p>
--

Очевидно, что результаты работы немного отличаются, хотя графы одинаковы. Это связано с эвристической функцией, она зависит от названий вершин, а не от расстояния, как должна. Также эвристическая функция не является монотонной, так как есть контрпример: пусть конечная вершина графа  $Z$ , а в нее можно попасть из  $A$ . Тогда,  $h(Z) - h(A) = 25$  по условию задания, а фактический вес ребра не исключает меньшего значения, что противоречит условию монотонности. Но тем не менее ответы все равно сошлись.

### **Вывод.**

В ходе работы были реализованы два алгоритма, находящие минимальный путь в графе: жадный и  $A^*$ . Жадный алгоритм находит не самый короткий путь, так как в нем выбирается локально минимальная дуга.  $A^*$  в свою очередь находит минимальный путь так как при его использовании хранится длина пути от стартовой до текущей. Характерной чертой алгоритма  $A^*$  является



эвристическая функция, которая оказывает влияние на приоритет выбора вершины для рассмотрения. При использовании корректной эвристической функции можно добиться уменьшения временной сложности, при некорректной не только можно увеличить время выполнения, но и получать некорректный результат. В данной лабораторной работе эвристика не является корректной, так как она не всегда монотонна.

## ПРИЛОЖЕНИЕ А

### Исходный код жадного алгоритма со стека.

```
from sys import stdin
from math import inf
class Edge:
    end = ''
    weight = 0
    def __init__(self, e, w):
        self.end = e
        self.weight = w
    def __str__(self):
        return str(self.end) + " " + str(self.weight)
    def __lt__(self, other):
        return (self.weight < other.weight)
    def __le__(self, other):
        return (self.weight <= other.weight)
if __name__ == "__main__":
    start, finish = input().split()
    # чтение графа
    edges = {}
    for line in stdin:
        u,v,c = line.replace('\n', '').split()
        if (u in edges.keys()):
            edges[u].append(Edge(v, float(c)))
        else:
            edges[u] = [Edge(v, float(c))]
    way = start
    cur = start
    while cur != finish:
        if cur in edges.keys():
            # поиск минимального пути из текущей вершины
            minimum = min(edges[cur])
            # проверка на повторное посещение
            if minimum.weight != inf:
                print("from",cur,"to",minimum.end, "because minimal cost of",
                    [(w.end, w.weight) for w in edges[cur]], "is", minimum.weight)
                minimum.weight = inf
                way += minimum.end
                cur = minimum.end
            # если минимальный сосед является уже посещенным, то значит тупик
            # необходимо удалить вершину из пути
            else:
                try:
                    print("Come back from", way[-1], "to", way[-2], "because all
neighbours is visited.")
                    way = way[:-1]
                    cur = way[-1]
                except IndexError:
                    print("No way.")
                    exit()
            # удаляем вершину, если из вершины некуда идти
            else:
                try:
                    print("Come back from", way[-1], "to", way[-2], "because here is
a dead end.")
                    way = way[:-1]
                    cur = way[-1]
                except IndexError:
                    print("No way.")
                    exit()
    print(way)
```

## ПРИЛОЖЕНИЕ Б

Исходный код A\* для задания на степике.

```
from sys import stdin
from math import inf

def h(u, finish):
    return abs(ord(finish) - ord(u))

def recPath(fset, start, finish):
    path = []
    curr = finish
    path.append(curr)
    while curr != start:
        curr = fromset[curr]
        path.append(curr)
    return "".join(reversed(path))

if __name__ == "__main__":
    start, finish = input().split()
    graph = {}
    # чтение графа
    for line in stdin:
        u,v,c = line.replace('\n', '').split()
        if u not in graph.keys():
            graph[u] = {v:float(c)} # begin end costEdge costFromStart
        else:
            graph[u][v] = float(c)
    # Алгоритм A*
    # инициализируем вспомогательные множества и словари
    closedset = [] # посещенные вершины
    openset = [] # раскрытые для рассмотрения вершины
    openset.append(start)
    print(start,"append START to openset")
    print("openset:", openset)
    fromset = {} # словарь оптимальных переходов
    g = {} # стоимости путей из начальной вершины
    g[start] = 0
    f = {} # = стоимость пути из начала плюс значение эвристической
    функции
    f[start] = g[start] + h(start, finish)
    # пока множество раскрытых вершин не опустеет
    while len(openset) > 0:
        # поиск минимального по f
        curr = openset[0]
        for u in openset:
            if f[curr] > f[u]:
                curr = u
        print("Take minimal f(",curr,") =", f[curr], "from", f)
        # предварительный выход
        if curr == finish:
            break
        # извлекаем рассмотренную вершину и добавляем ее в посещенные
        openset.pop(openset.index(curr))
        print(curr, "pop from openset")
        print("openset:", openset)
        closedset.append(curr)
        print(curr, "append to closedset")
        print("closedset:", closedset)
        if (curr in graph.keys()):
            # просмотр соседей
```

```

    for neib in graph[curr]:
        if neib in closedset:
            continue # если сосед просмотрен, то переход к
следующему
            tentScore = g[curr] + graph[curr][neib] # ожидаемый
кратчайший путь из начала
            if neib not in openset:
                openset.append(neib) # добавление соседа в раскрытые
вершины если его там нет
                print(neib, "neighbour of", curr, "append to openset")
                print("openset:", openset)
                tentIsBetter = True # если соседа не рассматривали, то
естественно путь стал лучше
            else:
                tentIsBetter = True if tentScore < g[neib] else False
# если рассматривали то нужно разобраться -- стал путь короче или нет
                if tentIsBetter: # если путь до соседа стал короче то
добавляем ребро в соответствующий словарь
                    fromset[neib] = curr
                    print("To", neib, "from", curr)
                    print("fromset", fromset)
                    g[neib] = tentScore # и обновляем словари g и f
                    f[neib] = g[neib] + h(neib, finish)
            print(" ".join(map(str, recPath(fromset, start, finish)))) #
восстанавливаем путь пользуясь тривиальной функцией

```

## ПРИЛОЖЕНИЕ В

Исходный код А\* со спецификациями по варианту.

```
#Terekhov_8382_v#1
from sys import stdin
from math import inf

def h(u, finish):
    return abs(finish - u)

def recPath(fset, start, finish):
    path = []
    curr = finish
    path.insert(0, curr)
    while curr != start:
        curr = fromset[curr]
        path.insert(0, curr)
    return path

if __name__ == "__main__":
    start, finish = map(int, input().split())
    graph = {}
    # чтение графа
    for line in stdin:
        u,v,c = line.replace('\n', '').split()
        if int(u) not in graph.keys():
            graph[int(u)] = {int(v):float(c)}
        else:
            graph[int(u)][int(v)] = float(c)
    # Алгоритм А*
    # инициализируем вспомогательные множества и словари
    closedset = [] # посещенные вершины
    openset = [] # раскрытые для рассмотрения вершины
    openset.append(start)
    print(start,"append START to openset")
    print("openset:", openset)
    fromset = {} # словарь оптимальных переходов
    g = {} # стоимости путей из начальной вершины
    g[start] = 0
    f = {} # = стоимость пути из начала плюс значение эвристической
    функции
    f[start] = g[start] + h(start, finish)
    # пока множество раскрытых вершин не опустеет
    while len(openset) > 0:
        # поиск минимального по f
        curr = openset[0]
        for u in openset:
            if f[curr] > f[u]:
                curr = u
        print("Take minimal f(",curr,") =", f[curr], "from", f)
        # предварительный выход
        if curr == finish:
            break
        # извлекаем рассмотренную вершину и добавляем ее в посещенные
        openset.pop(openset.index(curr))
        print(curr, "pop from openset")
        print("openset:", openset)
        closedset.append(curr)
        print(curr, "append to closedset")
        print("closedset:", closedset)
        if (curr in graph.keys()):
            # просмотр соседей
```

```

        for neib in graph[curr]:
            if neib in closedset:
                continue # если сосед просмотрен, то переход к
следующему
                tentScore = g[curr] + graph[curr][neib] # ожидаемый
кратчайший путь из начала
                if neib not in openset:
                    openset.append(neib) # добавление соседа в раскрытые
вершины если его там нет
                    print(neib, "neighbour of", curr, "append to openset")
                    print("openset:", openset)
                    tentIsBetter = True # если соседа не рассматривали, то
естественно путь стал лучше
                else:
                    tentIsBetter = True if tentScore < g[neib] else False
# если рассматривали то нужно разобраться -- стал путь короче или нет
                    if tentIsBetter: # если путь до соседа стал короче то
добавляем ребро в соответствующий словарь
                        fromset[neib] = curr
                        print("To", neib, "from", curr)
                        print("fromset", fromset)
                        g[neib] = tentScore # и обновляем словари g и f
                        f[neib] = g[neib] + h(neib, finish)
                print(" ".join(map(str, recPath(fromset, start, finish)))) #
восстанавливаем путь пользуясь тривиальной функцией

```