

**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: Алгоритмы поиска пути в графах**

Студент гр. 8382

Щемель Д.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

## Задание

### Жадный алгоритм

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
```

### A\*

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
```

Вариант No 4. Модификация  $A^*$  с двумя финишами (требуется найти путь до любого из двух).

## Описание алгоритма

### А\*

Сначала происходит считывание данных графа: начальная и конечные вершины, пути. Все данные сохраняются в объекте самого графа. После этого вызывается функция нахождения лучшего пути из начальной вершины до ближайшей конечной. Функция работает следующим образом: сначала выбирается начальная вершина. Для всех вершин до которых можно добраться из текущей выставляется минимальный путь и записывается вершина, из которой до данной можно добраться по минимальному пути. Для каждой вершины рассчитывается приоритет - сумма минимального пути до вершины и значение эвристической функции - разница между символами в таблице ASCII. Вершина добавляется в словарь по ключу-приоритету(аналог очереди с приоритетом). На следующей итерации выбирается вершина с минимальным приоритетом. Если эта вершина - конечная, тогда алгоритм завершает работу.

После завершения поиска минимального пути - выводится минимальный путь до этой вершины, который был сохранён в вершинах.

### Жадный алгоритм

Процесс считывания аналогичен тому, что в А\*. Разница заключается в том, что на каждой итерации выбирается вершина, расстояние до которой от текущей минимально.

### Сложности алгоритма

По памяти:  $O(N^2)$  для хранения вершин и путей - для обоих алгоритмов.

По количеству операций: А\* -  $O(N^2)$ , потому что для каждой вершины мы обходим её соседей. И в худшем случае (при выборе не оптимальной эвристики) нам придётся обойти весь граф + поиск по приоритету. Для жадного -  $O(N^2)$ , где N - количество вершин.

## Описание функций и структур данных

Используемые классы:

- *class Graph* - для хранения вершин графа. А так же начальной и конечных вершин.
- *class Node* - для хранения информации о вершине: Имя, минимальное расстояние до неё и ссылка на родительскую вершину в минимальном пути до неё. А так же пути до других вершин.

Используемые функции: - *static Graph ReadGraph()* - считывает входные данные и возвращает на основе них граф. - *static string FindBestWay(Graph graph)* - ищет минимальный путь в графе от начальной вершины до одной из конечных. Возвращает минимальный путь в виде строки.

## Тестирование

A\*

Input	Output
a e	Current node => a
a b 3.0	Add node => b with priority 6
b c 1.0	Add node => d with priority 6
c d 1.0	Current node => b
a d 5.0	Add node => c with priority 6
d e 1.0	Current node => d
	Add node => e with priority 6
	Current node => c
	Add node => d with priority 6
	Current node => e
	ade
a e c	abc
a b 3.0	
b c 1.0	
c d 1.0	
a d 5.0	
d e 1.0	
a e	abde
a e 5.0	
a d 4.0	
d e 1.0	
a b 1.0	
b d 1.0	

Жадный алгоритм\*

Input	Output
a e	ade
a b 3.0	
b c 1.0	
c d 1.0	
a d 5.0	
d e 1.0	
a e	abc
a b 3.0	
b c 1.0	
c d 1.0	
a d 5.0	
d e 1.0	
a e	abde
a e 5.0	
a d 4.0	
d e 1.0	
a b 1.0	
b d 1.0	
a e	Next way => b
a b 3.0	Next way => c
b c 1.0	Next way => d
c d 1.0	Next way => d
a d 5.0	Next way => e
d e 1.0	abcde

## **Вывод**

В ходе выполнения лабораторной работы были получены практические навыки применения алгоритмов на графах, а именно: жадный алгоритм и алгоритм A\*.

## Приложение А. Исходный код

```
#define DEBUG

using System;
using System.Collections.Generic;
using System.Linq;

namespace lab2
{
    /// <summary>
    /// Class for storing start and end points of graph
    /// </summary>
    public class Graph
    {
        /// <summary>
        /// Node in graph
        /// </summary>
        public class Node
        {
            /// <summary>
            /// Name of node
            /// </summary>
            public char Name { get; set; }

            /// <summary>
            /// Min distance to node
            /// </summary>
            public double? MinDistance { get; set; }

            /// <summary>
            /// Previous node in best way to node
            /// </summary>
            public Node CameFrom { get; set; }

            /// <summary>
            /// Children to children nodes
            /// </summary>
            public Dictionary<Node, double> Children { get; } = new Dictionary<Node, double>();
        }

        /// <summary>
        /// Start point
    }
}
```



```

    /// </summary>
    public Node Start { get; set; }

    /// <summary>
    /// End point
    /// </summary>
    public List<Node> End { get; set; } = new List<Node>();

    /// <summary>
    /// Map of nodes in graph by their names
    /// </summary>
    public Dictionary<char, Node> Nodes { get; } = new Dictionary<char, Node>();
}

```

```

class Program
{
    /// <summary>
    /// Read graph from stdin
    /// </summary>
    /// <returns>Graph</returns>
    static Graph ReadGraph()
    {
        var startEnd = Console.ReadLine().Split(' ');
        var graph = new Graph()
        {
            Start = new Graph.Node
            {
                Name = startEnd[0].First()
            },
        };

        for (var i = 1; i < startEnd.Length; ++i)
        {
            graph.End.Add(new Graph.Node {Name = startEnd[i].First()});
        }

        while (true)
        {
            var input = Console.ReadLine()?.Split(' ');
            if (input == null || input.Length != 3)
            {
                break;
            }
        }
    }
}

```

```

    }

    var newNodeStart = new Graph.Node() {Name = input[0].First()};
    var newNodeEnd = new Graph.Node() {Name = input[1].First()};
    var distance = double.Parse(input[2]);

    Graph.Node existingStartNode = null;

    if (!graph.Nodes.TryGetValue(newNodeStart.Name, out existingStartNode))
    {
        graph.Nodes.Add(newNodeStart.Name, newNodeStart);
        existingStartNode = newNodeStart;
    }

    Graph.Node existingEndNode = null;
    if (!graph.Nodes.TryGetValue(newNodeEnd.Name, out existingEndNode))
    {
        graph.Nodes.Add(newNodeEnd.Name, newNodeEnd);
        existingEndNode = newNodeEnd;
    }

    existingStartNode.Children.Add(existingEndNode, distance);
}

graph.Start = graph.Nodes[graph.Start.Name];

for (var i = 0; i < graph.End.Count; ++i)
{
    graph.End[i] = graph.Nodes[graph.End[i].Name];
}

return graph;
}

/// <summary>
/// Find greed way in graph
/// </summary>
/// <param name="graph">Graph for find.</param>
/// <returns>Greed way to end node in graph as string</returns>
static string FindGreedWay(Graph graph)
{
    var visitedNodes = new Stack<Graph.Node>();

```

```

visitedNodes.Push(graph.Start);

while (visitedNodes.Count != 0)
{
    var tmp = visitedNodes.First();
    while (tmp.Children.Count != 0)
    {
        tmp = tmp.Children.OrderBy(x => x.Value).First().Key;
        #if (DEBUG)
        Console.WriteLine($"Next way =>{tmp}");
        #endif
        visitedNodes.Push(tmp);

        if (graph.End.Contains(tmp))
        {
            return string.Join("", visitedNodes.Reverse().Select(x => x.Name));
        }
    }

    visitedNodes.Pop();
    var last = visitedNodes.Peek();
    if (last.Children.Count != 0)
    {
        last.Children.Remove(last.Children.OrderBy(x => x.Value).First().Key);
    }
}

return string.Empty;
}

/// <summary>
/// Find best way to end node in graph
/// </summary>
/// <param name="graph">Graph for find</param>
/// <returns>Best way to end node in graph as string</returns>
static string FindBestWay(Graph graph)
{
    var nodesToVisit = new SortedDictionary<double, Queue<Graph.Node>> {{0, new Queue<Graph.Node>()}};
    nodesToVisit[0].Enqueue(graph.Start);
    nodesToVisit.First().Value.First().MinDistance = 0;
}

```

```

while (nodesToVisit.Count != 0)
{
    var tmp = nodesToVisit.First().Value.Dequeue();
    #if (DEBUG)
    Console.WriteLine($"Current node =>{tmp.Name}");
    #endif
    if (nodesToVisit.First().Value.Count == 0)
    {
        nodesToVisit.Remove(nodesToVisit.First().Key);
    }

    if (graph.End.Contains(tmp))
    {
        var bestWay = new Stack<Graph.Node>();
        bestWay.Push(tmp);
        while (tmp.CameFrom != null)
        {
            bestWay.Push(tmp.CameFrom);
            tmp = tmp.CameFrom;
        }

        return string.Join("", bestWay.Select(x => x.Name));
    }

    foreach (var way in tmp.Children)
    {
        var distance = tmp.MinDistance + way.Value ?? 0;
        var heuristic = distance + Math.Abs(graph.End.Min(x => x.Name) - way.Key.Name);

        if (!nodesToVisit.ContainsKey(heuristic))
        {
            nodesToVisit[heuristic] = new Queue<Graph.Node>();
        }

        nodesToVisit[heuristic].Enqueue(way.Key);
        #if (DEBUG)
        Console.WriteLine($"Add node =>{way.Key.Name} with priority {heuristic}");
        #endif
        if (way.Key.MinDistance == null || distance < way.Key.MinDistance)
        {

```

```

        way.Key.MinDistance = distance;
        way.Key.CameFrom = tmp;
    }
}

return string.Empty;
}

static void Main(string[] args)
{
    var graph = ReadGraph();
    Console.WriteLine(FindBestWay(graph)); // For greed way replace with FindGreedWay
}
}

```