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

# ОТЧЕТ

# по лабораторной работе №3

по дисциплине «Алгоритмы и Структуры Данных»

**Тема: «Алгоритмы на графах» Вариант 1** 

Студент гр. 8301	 Забалуев Д.А
Преподаватель	 Тутуева А.В.

Санкт-Петербург

2020

# 1. Цель работы

Реализовать алгоритм Дейкстры на графе, представляемом при помощи списков смежности. А также найти наиболее эффективный по стоимости перелет из города i в город j, используя список возможных авиарейсов, данный в текстовом формате в .txt файле.

## 2. Описание программы

Идея алгоритма состоит в следующем:

Каждой вершине V графа G сопоставляется метка — минимальное известное расстояние от этой вершины до стартовой вершины S. Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

Метка самой вершины S полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от S до других вершин пока неизвестны. Все вершины графа помечаются как непосещённые.

Если все вершины посещены, алгоритм завершается. В противном случае, из ещё не посещённых вершин выбирается вершина V, имеющая минимальную метку. Рассматриваются всевозможные маршруты, в которых V является предпоследним пунктом. Для каждого соседа вершины V, кроме отмеченных как посещённые, рассматривается новая длина пути, равная сумме значений текущей метки V и длины ребра, соединяющего V с этим соседом. Если полученное значение длины меньше значения метки соседа, значение метки заменяется полученным значением длины. Рассмотрев всех соседей, вершина V помечается как посещённая и шаг алгоритма повторяется.

В данной программе используются следующие структуры данных:

### • Класс Graph:

Данный класс реализует граф, основанный на списках смежности. Этот класс хранит: список смежности adjList (динамический массив из контейнеров Мар), который помимо номера смежной вершины хранит ещё и вес ребра; список namesList, являющийся связью между названием города и номером вершины; очередь с приоритетами markList, которая является связью между номером вершины, её меткой и информацией о том, посещена вершина или нет.

Конструктор данного класса получает в качестве параметра список названий городов и на его основе инициализирует остальные поля класса.

Данный класс имеет два публичных метода – createFrom(...) и dijkstra(...). createFrom строит граф (заполняет списки смежности) на основе информации из текстового файла; dijkstra(...) реализует сам алгоритм Дейкстры, возвращая наиболее дешевый маршрут в виде строки.

Данный класс имеет один приватный метод – buildPath(...), который строит самый дешевый путь из размеченного после обхода по Дейкстре графа.

### • Класс List:

Необычный шаблонный двусвязный список. Функционал списка дополнен возможностью проверки наличия элемента в списке, возможностью получения индекса определенного элемента, а также реализованы конструкторы перемещения и копирования и перегружены соответствующие им операторы присвоения для повышения гибкости списка. Используется для хранения введенной информации, списка названий городов, для хранения результирующего маршрута и ещё много чего.

• Класс Мар:

Обычный шаблонный ассоциативный массив. Используется в качестве списка смежности для одновременного хранения смежной вершины и веса ребра, направленного к этой вершине.

Класс priority\_queue:

Необычная шаблонная очередь с приоритетами на основе двоичной minHeap. Функционал очереди дополнен возможностью получать приоритет элементов очереди, а также получать элемент с минимальным приоритетом, не удаляя его из очереди. Используется для хранения меток вершин. Приоритет элементов выступает в качестве метки, а наличие элемента в очереди — в качестве информации о том, была посещена вершина или нет.

# 3. Оценка временной сложности методов

N – количество строк входной информации

М – количество символов в строке

V – количество вершин (городов)

Е – количество ребер (возможных перелетов между городами)

- 1) *convert* имеет временную сложность O(N\*M)
- 2)  $get\_unique\_names$  имеет временную сложность O(N\*V); N\*(V\*1 + V\*1) = N\*V
- 3) *Graph* имеет временную сложность O(V);
- 4) *createFrom* имеет временную сложность  $O(N^2 + NV + N*log(E))$ ;  $N*(N + V + V + 2*log(E)) = N^2 + NV + N*log(E)$
- 5) **buildPath** имеет временную сложность  $O(VE+E^3)$ ; log(E)+log(E)+V+2V+3V+V+E(E+E(E+log(E)+2V)+V)=

$$= log(E) + V + E^2 + VE + E^3 + ElogE + EV = EV + E*log(E) + E^3$$

6) dijkstra имеет временную сложность  $O(E^3+V^2*E*log(V)+V*E*log(E)+V*E^2+V^2*E);$   $2V+V*log(V)+V(V+log(V)+E+E(E+V+log(E)+V+V*log(V))+log(V))+VE+E^3+2V=$   $=E^3+V^2*E*log(V)+V*E*log(E)+V*E^2+V^2*E$ 

# 4. Примеры работы

Saint-Petersburg; Moscow; 10; 20; Moscow; Habarovsk; 40; 35; Saint-Petersburg; Habarovsk; 14; N/A; Vladivostok; Habarovsk; 13; 8; Vladivostok; Saint-Petersburg; N/A; 20; City 1: Vladivostok

City 2: Saint-Petersburg

The cheapest route: Vladivostok -> Habarovsk -> Moscow -> Saint-Petersburg Total cost: 68

```
City 1: Moscow
Saint-Petersburg; Moscow; 10; 20;
                                                City 2: Tokyo
Moscow; Habarovsk; 40; 35;
Saint-Petersburg; Habarovsk; 14; N/A;
Vladivostok; Habarovsk; 13;8;
Vladivostok; Saint-Petersburg; N/A; 20;
                                                Irrelevant start or end city name!
                                         City 1: Habarovsk
Saint-Petersburg; Moscow; 10; 20;
                                         City 2: Habarovsk
Moscow; Habarovsk; 40; 35;
Saint-Petersburg; Habarovsk; 14; N/A;
Vladivostok; Habarovsk; 13;8;
Vladivostok; Saint-Petersburg; N/A; 20;
                                         Start and destination points must be different!
                                     City 1: Moscow
                                     City 2: Saint-Petersburg
 Saint-Petersburg; Moscow; 10; N/A;
 Moscow; Habarovsk; 40; N/A;
                                     There is no route between Moscow and Saint-Petersburg
      SPB;MSC;10;20;
      MSC;HBR;15;35;
                                          City 1: PPK
      HBR; SPB; N/A; 45;
      VDK; HBR; 15; 20;
                                          City 2: SPB
      SPB; VDK; 20; N/A;
      PPK;HBR;15;10;
      VDK; PPK; N/A; 5;
                                          The cheapest route: PPK -> HBR -> MSC -> SPB
      TKY; MSC; 40; 60;
                                          Total cost: 70
      PPK;TKY;20;15;
      PKN;TKY;50;N/A;
      SPB;MSC;10;20;
      MSC;HBR;15;35;
                                          City 1: MSC
      HBR; SPB; N/A; 45;
      VDK; HBR; 15; 20;
                                          City 2: TKY
      SPB; VDK; 20; N/A;
      PPK;HBR;15;10;
      VDK; PPK; N/A; 5;
                                          The cheapest route: MSC -> HBR -> PPK -> TKY
      TKY;MSC;40;60;
                                          Total cost: 45
      PPK;TKY;20;15;
      PKN;TKY;50;N/A;
```

### 5. Листинг

```
MAIN.CPP
```

```
#include <iostream>
#include <fstream>
#include "dijkstra_algorithm.h"
int main()
{
       std::ifstream file("info.txt");
       List<std::string> lines;
       while(file)
               std::string str;
               getline(file, str);
               lines.push_back(str);
       lines.pop_back();
       List<std::string*> normalized_info = convert(lines);
       List<std::string> namesList = get_unique_names(normalized_info);
       Graph G(namesList);
       G.createFrom(normalized_info);
       std::string start, end;
       cout << "City 1: ";</pre>
       cin >> start;
       cout << "\nCity 2: ";</pre>
       cin >> end;
       cout << endl << endl;</pre>
       cout << G.dijkstra(start, end);</pre>
       cout << endl << endl;</pre>
       for (size_t i = 0; i < normalized_info.getSize(); ++i)</pre>
               delete[] normalized_info.at(i);
       }
       return 0;
}
```

### HUFFMANALGORITHM.H

```
#pragma once
#include "Queue.h"
#include "List.h"
```

```
#include "Map.h"
#include "priority_queue.h"
#include <string>
#include <stdexcept>
constexpr uint8_t CITY_FROM = 0;
constexpr uint8_t CITY_TO = 1;
constexpr uint8_t COST_FORWARD = 2;
constexpr uint8_t COST_BACKWARD = 3;
constexpr uint8_t DATA_INPUT_WORDS_AMOUNT = 4;
template<class T>
List<T> reverse(List<T> lst);
class Graph
{
private:
      Map<uint16_t, uint64_t>* adjList; //adjacency list that stores vertices number and edge weight
       List<std::string> namesList; //represents link between city name and vertices number
       priority queue<uint16 t> markList; //stores vertices number and it's mark
       size_t size;
       //creates from start point to the destination point using marked graph
       List<std::string> buildPath(priority_queue<uint16_t>& visitedMarksList, const uint16_t& start,
const uint16_t& destination)
       {
             List<std::string> path;
             //checks if the straight route from start to destination is the cheapest
             if (adjList[start].contains(destination))
                    if (adjList[start].find(destination) == visitedMarksList.get_priority(destination))
                           path.push_front(namesList.at(destination));
                           path.push_front(namesList.at(start));
                           return path;
                    }
             }
             //checks if destination point is reachable from the start
             if (visitedMarksList.get_priority(destination) == UINT64_MAX)
                    path.push_front("There is no route between " + namesList.at(start) + " and " +
namesList.at(destination));
                    return path;
             path.push front(namesList.at(destination));
             auto current = destination;
             auto cur mark = visitedMarksList.get priority(destination);
             //the algorithm makes path from the destination point to the start
             while(current != start)
             {
                    std::string city;
                    //getting all current's neighbors
                    auto neighbors_list = adjList[current].get_keys();
                    uint16_t neighbor;
                    //checks every neighbor's mark if (current's mark - edge weight) equals it
                    for(size_t i = 0; i < neighbors_list.getSize(); ++i)</pre>
                    {
                           neighbor = neighbors_list.at(i);
```

```
if(cur mark - adjList[neighbor].find(current) ==
visitedMarksList.get_priority(neighbor))
                                   city = namesList.at(neighbor);
                                  break;
                            }
                     }
                    path.push_front(city);
                    current = neighbor;
                     cur_mark = visitedMarksList.get_priority(neighbor);
              }
              return path;
       }
public:
       //default graph initializing constructor
       explicit Graph(List<std::string>& namesList)
              : size(namesList.getSize())
       {
              this->namesList = namesList;
              adjList = new Map<uint16_t, uint64_t>[size];
              for (size_t i = 0; i < size; ++i)</pre>
                    markList.insert(i, UINT64_MAX);
       }
       ~Graph()
              delete[] adjList;
       }
       //creates graph from list of words arrays
      void createFrom(List<std::string*>& info) const
              for (size_t i= 0; i < info.getSize(); ++i)</pre>
              {
                    const auto line = info.at(i);
                    uint16_t city_from_number = namesList.find(line[CITY_FROM]);
                    uint16_t city_to_number = namesList.find(line[CITY_TO]);
                    uint64 t cost forward = std::stoull(line[COST FORWARD]);
                    uint64_t cost_backward = std::stoull(line[COST_BACKWARD]);
                     //filling the adjacency list from input info
                    for(auto j = 1; j <= 2; ++j)
                     {
                           adjList[city from number].insert(city to number, cost forward);
                           std::swap(city_from_number, city_to_number);
                           std::swap(cost_forward, cost_backward);
                     }
              }
       }
       //finding the cheapest/shortest way with dijkstra algorithm
       std::string dijkstra(const std::string& start_point,const std::string& destination_point)
              if (start_point == destination_point)
                     return "Start and destination points must be different!";
              if (!namesList.contains(start_point) || !namesList.contains(destination_point))
                    return "Irrelevant start or end city name!";
```

```
priority queue<uint16 t> visitedMarksList; //stores visited vertices with their final
marks
              const uint16_t start = namesList.find(start_point); //beginning of the required path
              const uint16_t destination = namesList.find(destination_point); //end of the required path
             markList.update(start, 0);
             while (markList.getSize())
              {
                    const auto current_mark = markList.get_priority();
                    const auto current = markList.extract_min(); // vertices that is currently being
processed
                    const auto neighbors_list = adjList[current].get_keys(); //getting all current's
neighbors
                    for (size_t j = 0; j < neighbors_list.getSize(); ++j)</pre>
                           const auto neighbor = neighbors_list.at(j); //getting current being processed
neighbor
                           const auto visited = !markList.contains(neighbor); //find out if this
neighbor has been already visited
                           const auto edge weight = adjList[current].find(neighbor);
                           if(!visited && edge_weight !=0)
                           {
                                   const auto mark = current_mark + edge_weight;
                                  if (markList.get_priority(neighbor) > mark)
                                         markList.update(neighbor, mark);
                           }
                    }
                    visitedMarksList.insert(current, current_mark);
              }
              //getting route from marked graph
              const auto way = buildPath(visitedMarksList, start, destination);
              std::string result;
              if (way.getSize() > 1)
                    result = "The cheapest route: ";
                    for (size_t i = 0; i < way.getSize() - 1; ++i)</pre>
                           result += way.at(i) + " -> ";
                    result += way.at(way.getSize() - 1) + "\nTotal cost: " +
std::to string(visitedMarksList.get priority(destination));
             }
              else
                    result = way.at(0);
              return result;
       }
};
//converts input list of lines into list of words arrays
inline List<std::string*> convert(List<std::string>& info)
{
       if (info.getSize() == 0)
             throw std::length_error("Input was empty!");
      List<std::string*> separated_info;
      for (size_t i = 0; i < info.getSize(); ++i)</pre>
```

```
{
              const auto line = new std::string[DATA INPUT WORDS AMOUNT];
              auto word number = 0;
              for (auto ch : info.at(i))
                    if (ch == ';')
                           word_number++;
                    else
                           line[word_number] += ch;
              }
              if (word_number > DATA_INPUT_WORDS_AMOUNT)
                     throw std::length_error("Irrelevant input format!");
              if (line[COST FORWARD] == "N/A")
                     line[COST_FORWARD] = "0";
              if (line[COST BACKWARD] == "N/A")
                     line[COST_BACKWARD] = "0";
              if (line[COST FORWARD] == "0" && line[COST BACKWARD] == "0")
                     throw std::logic_error("Flight was unavailable in both directions!");
              separated_info.push_back(line);
      }
       return separated_info;
}
//returns list of all cities names (without repeats)
inline List<std::string> get_unique_names(List<std::string*>& info)
      List<std::string> namesList;
      for (size_t i = 0; i < info.getSize(); ++i)</pre>
              if (!namesList.contains(info.at(i)[CITY_FROM]))
                    namesList.push_back(info.at(i)[CITY_FROM]);
              if (!namesList.contains(info.at(i)[CITY_T0]))
                    namesList.push_back(info.at(i)[CITY_T0]);
      }
       return namesList;
}
template<class T>
List<T> reverse(List<T> lst)
{
      List<T> reversed;
       for (size t i = 1; i <= lst.getSize(); ++i)</pre>
              reversed.push_back(lst.at(lst.getSize() - i));
       return reversed;
}
```