**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

## 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.

## 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 Map), 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 Map:

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.

## Оценка временной сложности методов

*N – количество строк входной информации*

*M – количество символов в строке*

*V – количество вершин (городов)*

*E – количество ребер (возможных перелетов между городами)*

1. ***convert*** имеет временную сложность О(*N\*M*)
2. ***get\_unique\_names*** имеет временную сложность О(*N\*V*);

*N\*(V\*1 + V\*1) = N\*V*

1. ***Graph*** имеет временную сложность О(*V*);
2. ***createFrom*** имеет временную сложность О(*N² + NV + N\*log(E)*);

*N\*(N + V + V + 2\*log(E)) = N² + NV + N\*log(E)*

1. ***buildPath*** имеет временную сложность О(*VE+E³*);

*log(E)+log(E)+V+2V+3V+V+E(E+E(E+log(E)+2V)+V)=   
= log(E)+V+E²+VE+ E³ + ElogE +EV = EV+E\*log(E)+ E³*

1. ***dijkstra*** имеет временную сложность О(*V²+E³+V²\*E\*log(V)+V\*E\*log(E)+V\*E²+V²\*E*);

*2V+V\*log(V)+V(V+log(V)+E+E(E+V+log(E)+V+V\*log(V))+log(V))+VE+ E³+2V=*

= *V²+E³+V²\*E\*log(V)+V\*E\*log(E)+V\*E²+V²\*E*

## Примеры работы

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## 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: ";

cin >> start;

cout << "\nCity 2: ";

cin >> end;

cout << endl << endl;

cout << G.dijkstra(start, end);

cout << endl << endl;

for (size\_t i = 0; i < normalized\_info.getSize(); ++i)

{

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)

{

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)

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)

{

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)

{

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)

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)

{

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)

{

if (!namesList.contains(info.at(i)[CITY\_FROM]))

namesList.push\_back(info.at(i)[CITY\_FROM]);

if (!namesList.contains(info.at(i)[CITY\_TO]))

namesList.push\_back(info.at(i)[CITY\_TO]);

}

return namesList;

}

template<class T>

List<T> reverse(List<T> lst)

{

List<T> reversed;

for (size\_t i = 1; i <= lst.getSize(); ++i)

reversed.push\_back(lst.at(lst.getSize() - i));

return reversed;

}