**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: нахождение кратчайшего пути по алгоритму Беллмана-Форда

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Касаткин А.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2022

Задача: найти наиболее эффективный по стоимости перелет из города i в город j используя алгоритм Беллмана-Форда.

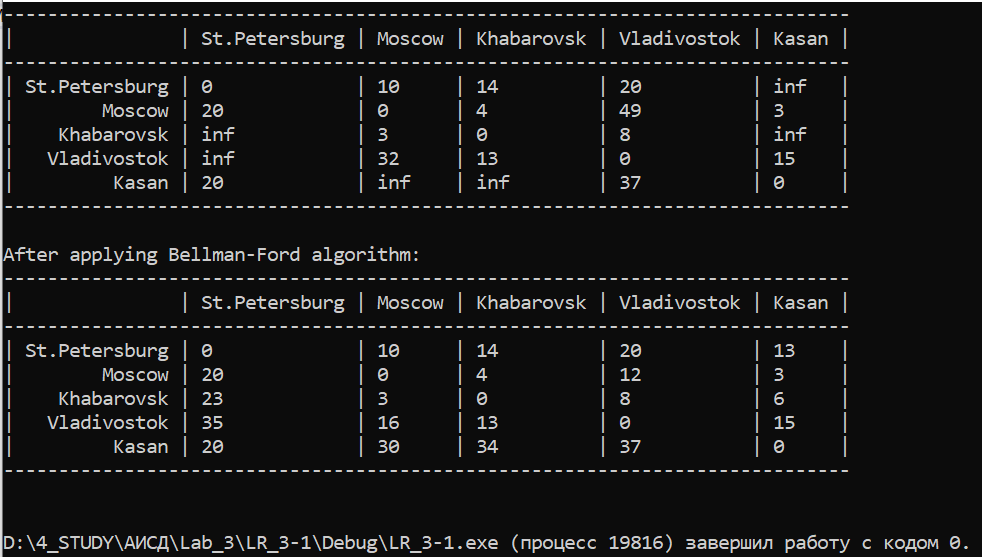
Описание алгоритмов:

Алгоритм позволяет найти кратчайший путь в ориентированном графе из исходной вершины до любой другой вершины этого графа.

Оценка временной сложности:

Ford\_Bellman O(n^4)

Результат:



Листинг:

**Main.cpp**

#include "FB.h"

int main()

{

setlocale(LC\_ALL, "Russian");

AdjMatrix graph;

ifstream fin;

const char filename[] = "input.txt";

fin.open(filename);

if (fin.good()) fin >> graph;

else return 5;

cout << "Initial graph's adjecent matrix is\n" << graph << "\n";

cout << "After applying Bellman-Ford algorithm:\n" << graph.FordBellman() << endl;

fin.close();

return 0;

}

**FB.h**

#include "linear.h"

#include <fstream>

#include <iomanip>

#define INF ((double)(1e300 \* 1e300))

const string NOT\_AVAILABLE = "N/A";

class AdjMatrix

{

size\_t size;

string\* names;

double\*\* values;

void outputline(ostream& stream, AdjMatrix& matrix, size\_t maxwidth)

{

stream << "--";

for (size\_t i = 0; i < maxwidth; i++) stream << '-';

stream << "---";

for (size\_t i = 0; i < matrix.size; i++)

{

const size\_t len = matrix.names[i].length();

for (size\_t j = 0; j < len; j++)

{

stream << '-';

}

stream << "--" << (i == matrix.size - 1 ? '\n' : '-');

}

}

public:

AdjMatrix() : values(nullptr), names(nullptr), size(0) {}

AdjMatrix(size\_t size, string\* names, double\*\* values) : values(values), names(names), size(size) {}

~AdjMatrix()

{

if (values != nullptr)

{

for (size\_t i = 0; i < size; i++)

{

delete[] values[i];

}

delete[] values;

}

}

friend bool operator==(AdjMatrix& gr1, AdjMatrix& gr2)

{

if (gr1.size != gr2.size) return false;

for (size\_t i = 0; i < gr1.size; i++) if (gr1.names[i] != gr2.names[i]) return false;

for (size\_t i = 0; i < gr1.size; i++)

{

for (size\_t j = 0; j < gr1.size; j++)

{

if (gr1.values[i][j] != gr2.values[i][j]) return false;

}

}

return true;

}

friend ifstream& operator>>(ifstream& stream, AdjMatrix& matrix)

{

string name1\_buffer, name2\_buffer;

double value1\_buffer, value2\_buffer;

List<string> names;

while (!stream.eof())

{

string temp;

stream >> name1\_buffer >> name2\_buffer >> value1\_buffer >> value2\_buffer;

if (names.Find(name1\_buffer) < 0) names.PushBack(name1\_buffer);

if (names.Find(name2\_buffer) < 0) names.PushBack(name2\_buffer);

if (value1\_buffer == 0)

{

stream.clear();

stream >> temp >> value2\_buffer;

}

if (value2\_buffer == 0)

{

stream.clear();

if (temp == NOT\_AVAILABLE) throw invalid\_argument("Wrong input");

stream >> temp;

}

}

matrix.size = names.GetSize();

matrix.names = new string[matrix.size];

matrix.values = new double\* [matrix.size];

for (size\_t i = 0; i < matrix.size; i++)

{

matrix.names[i] = names.GetData();

names.PopFront();

matrix.values[i] = new double[matrix.size];

for (size\_t j = 0; j < matrix.size; j++)

{

matrix.values[i][j] = (i == j ? 0 : INF);

}

}

stream.seekg(ios::beg);

while (!stream.eof())

{

string temp;

stream >> name1\_buffer >> name2\_buffer >> value1\_buffer >> value2\_buffer;

if (value1\_buffer == 0)

{

stream.clear();

stream >> temp >> value2\_buffer;

value1\_buffer = INF;

}

if (value2\_buffer == 0)

{

stream.clear();

stream >> temp;

value2\_buffer = INF;

}

size\_t i = 0, j = 0, k;

for (k = 0; k < matrix.size; k++)

{

if (matrix.names[k] == name1\_buffer) i = k;

if (matrix.names[k] == name2\_buffer) j = k;

}

matrix.values[i][j] = value1\_buffer;

matrix.values[j][i] = value2\_buffer;

}

return stream;

}

friend ostream& operator<<(ostream& stream, AdjMatrix& matrix)

{

size\_t maxwidth = 0;

for (size\_t i = 0; i < matrix.size; i++)

{

if (matrix.names[i].length() > maxwidth) maxwidth = matrix.names[i].length();

}

matrix.outputline(stream, matrix, maxwidth);

stream << "| ";

for (size\_t i = 0; i < maxwidth; i++)

{

stream << ' ';

}

stream << " | ";

for (size\_t i = 0; i < matrix.size; i++)

{

stream << setw(matrix.names[i].length()) << left << matrix.names[i] << " | ";

}

stream << "\n";

matrix.outputline(stream, matrix, maxwidth);

for (size\_t i = 0; i < matrix.size; i++)

{

stream << "| " << setw(maxwidth) << right << matrix.names[i] << " | ";

for (size\_t j = 0; j < matrix.size; j++)

{

stream << setw(matrix.names[j].length()) << left << matrix.values[i][j] << " | ";

}

stream << "\n";

}

matrix.outputline(stream, matrix, maxwidth);

return stream;

}

AdjMatrix& FordBellman()

{

if (size == 0) throw logic\_error("Adjecent matrix does not have a graph yet");

AdjMatrix\* newMatrix = new AdjMatrix();

newMatrix->size = size;

newMatrix->names = names;

newMatrix->values = new double\* [size];

// Choosing the vertex to calculate from

for (size\_t vertex = 0; vertex < size; vertex++)

{

// Initializing array with 0 on a current vertex and +INF on other vertexes

double\* distances = new double[size];

for (size\_t i = 0; i < size; i++)

{

distances[i] = (i == vertex ? 0 : INF);

}

// Repeating algorithm [vertex count - 1] times (enough)

for (size\_t step = 0; step < size; step++)

{

// Relaxing edges

for (size\_t i = 0; i < size; i++)

{

for (size\_t j = 0; j < size; j++)

{

if (i != j && values[i][j] < INF && distances[i] + values[i][j] < distances[j])

{

if (step == size - 1) throw logic\_error("Graph has negative cycles");

else distances[j] = distances[i] + values[i][j];

}

}

}

}

newMatrix->values[vertex] = distances;

}

return \*newMatrix;

}

inline double\*\* GetValues() { return values; }

};

**Linear.h**

#ifndef LINEAR\_STRUCT

#define LINEAR\_STRUCT

#include <iostream>

using namespace std;

template <class T>

class List

{

protected:

struct ListElement

{

ListElement\* previous, \* next;

T data;

ListElement(T& data, ListElement\* next = nullptr, ListElement\* previous = nullptr) : data(data), previous(previous), next(next) {}

~ListElement() {}

};

public:

ListElement\* head, \* tail;

List() : head(nullptr), tail(nullptr) {}

List(const size\_t size, T\* arr) : head(nullptr), tail(nullptr)

{

for (size\_t i = 0; i < size; i++) this->PushBack(arr[i]);

}

List(List<T>& ref) : head(nullptr), tail(nullptr)

{

const size\_t size = ref.GetSize();

for (size\_t i = 0; i < size; i++)

{

T newElem = ref.GetData(i);

this->PushBack(newElem);

}

}

template <typename... Args> List(Args... list) : head(nullptr), tail(nullptr) { this->PushBack(list...); }

template <typename... Args>

void PushBack(T first, Args&... rest)

{

this->PushBack(first);

this->PushBack(rest...);

}

~List()

{

if (head != nullptr) Clear();

}

List<T>& operator+(List<T>& l)

{

List<T>\* newList = new List<T>();

for (size\_t i = 0; i < GetSize(); i++) newList->PushBack(GetData(i));

for (size\_t i = 0; i < l.GetSize(); i++) newList->PushBack(l.GetData(i));

return \*newList;

}

void operator+=(List<T>& l)

{

\*this = \*this + l;

}

friend bool operator==(List<T>& list1, List<T>& list2)

{

ListElement\* current1 = list1.head;

ListElement\* current2 = list2.head;

while (current1 != nullptr)

{

if (current2 != nullptr)

{

if (current1->data != current2->data) return false;

}

else return false;

current1 = current1->next;

current2 = current2->next;

}

if (current2 != nullptr) return false;

return true;

}

friend ostream& operator<<(ostream& stream, List<T>& list)

{

ListElement\* current = list.head;

stream << "[";

while (current != nullptr)

{

stream << current->data;

if (current->next != nullptr) stream << " ";

current = current->next;

}

stream << "]";

return stream;

}

void Clear()

{

if (head == nullptr) throw logic\_error("List is already empty");

while (head != nullptr)

{

if (head->next == nullptr) break;

head = head->next;

delete head->previous;

}

delete head;

}

size\_t GetSize()

{

int size = 0;

ListElement\* current = head;

while (current != nullptr)

{

size++;

current = current->next;

}

return size;

}

int Find(T element)

{

ListElement\* current = head;

int index = 0;

while (current != nullptr)

{

if (current->data == element) return index;

current = current->next;

index++;

}

return -1;

}

virtual void PushFront(T element)

{

ListElement\* newElement = new ListElement(element, head);

if (head == nullptr)

{

head = newElement;

tail = head;

}

else

{

head->previous = newElement;

head = head->previous;

}

return;

}

virtual void PushBack(T element)

{

ListElement\* newElement = new ListElement(element, nullptr, tail);

if (tail == nullptr)

{

tail = newElement;

head = tail;

}

else

{

tail->next = newElement;

tail = tail->next;

}

return;

}

virtual void Push(T element)

{

throw logic\_error("Push is undefinable, use PushFront/PushBack/PushIndex");

}

virtual void PushIndex(int index, T element)

{

const size\_t lastIndex = GetSize() - 1;

if (index == 0) PushFront(element);

else if (index == lastIndex + 1) PushBack(element);

else if (index > lastIndex + 1 || index < 0) throw out\_of\_range("Stated index is invalid");

else

{

int currentIndex = 1;

ListElement\* current = head->next;

while (currentIndex != index)

{

current = current->next;

currentIndex++;

}

ListElement\* newElement = new ListElement(element, current, current->previous);

current->previous->next = newElement;

current->previous = newElement;

}

}

virtual void PopFront()

{

if (head == nullptr) throw logic\_error("List is already empty");

if (head->next == nullptr)

{

delete head;

head = nullptr;

tail = nullptr;

}

else

{

head = head->next;

delete head->previous;

head->previous = nullptr;

}

return;

}

virtual void PopBack()

{

if (tail == nullptr) throw logic\_error("List is already empty");

if (tail->previous == nullptr)

{

delete tail;

head = nullptr;

tail = nullptr;

}

else

{

tail = tail->previous;

delete tail->next;

tail->next = nullptr;

}

return;

}

virtual void PopIndex(int index)

{

const size\_t lastIndex = GetSize() - 1;

if (index == 0) PopFront();

else if (index == lastIndex) PopBack();

else if (index > lastIndex || index < 0) throw out\_of\_range("Stated index is invalid");

else

{

int currentIndex = 1;

ListElement\* current = head->next;

while (currentIndex != index)

{

current = current->next;

currentIndex++;

}

ListElement\* next = current->next, \* previous = current->previous;

delete current;

next->previous = previous;

previous->next = next;

}

}

virtual void Pop()

{

throw logic\_error("Push is undefinable, use PopFront/PopBack/PopIndex");

}

T& GetData(size\_t index = 0)

{

ListElement\* current = head;

size\_t i = 0;

while (current != nullptr)

{

if (i == index) return current->data;

i++;

current = current->next;

}

throw out\_of\_range("Index is incorrect");

}

inline void PrintData() { cout << \*this; }

};

template <typename T>

class Queue : public List<T>

{

public:

List<T>::Clear;

List<T>::GetSize;

void PushFront(T element) override

{

throw logic\_error("Queue structure has no access to execute PushFront");

}

List<T>::PushBack;

void PushIndex(int index, T element) override

{

if (index == GetSize()) PushBack(element);

else throw logic\_error("In queue structure pushing index can only be the last one");

return;

}

void Push(T element) override

{

PushBack(element);

return;

}

List<T>::PopFront;

void PopBack() override

{

throw logic\_error("Queue structure has no access to execute PopBack");

}

void PopIndex(int index) override

{

if (index == 0) PopFront();

else throw logic\_error("In queue structure popping index can only be zero");

return;

}

void Pop() override

{

PopFront();

return;

}

List<T>::GetData;

List<T>::PrintData;

};

template <typename T>

class Stack : public List<T>

{

public:

List<T>::Clear;

List<T>::GetSize;

List<T>::PushFront;

void PushBack(T element) override

{

throw logic\_error("Stack structure has no access to execute PushFront");

}

void PushIndex(int index, T element) override

{

if (index == 0) PushFront(element);

else throw logic\_error("In stack structure pushing index can only be zero");

return;

}

void Push(T element) override

{

PushFront(element);

return;

}

List<T>::PopFront;

void PopBack() override

{

throw logic\_error("Stack structure has no access to execute PopBack");

}

void PopIndex(int index) override

{

if (index == 0) PopFront();

else throw logic\_error("In stack structure popping index can only be zero");

return;

}

void Pop() override

{

PopFront();

return;

}

List<T>::GetData;

List<T>::PrintData;

};

#endif // LINEAR\_STRUCT