Балтийский государственный технический университет  
«ВОЕНМЕХ» им. Д. Ф. Устинова

Кафедра О7 «Информационные системы и программная инженерия»

**Практическая работа №2**по дисциплине «Структуры и организация данных»  
на тему «Нелинейные структуры данных»  
  
вариант 5

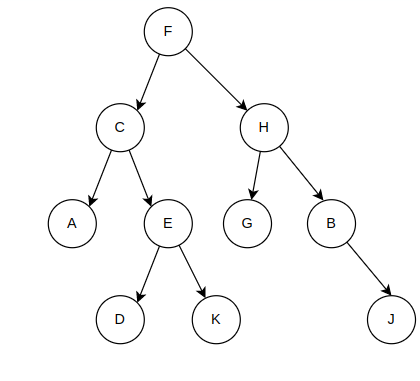
Выполнил:  
Студент Ковтун А. А  
Группа О716Б  
  
Преподаватель:  
Иванык А. О.

Санкт-Петербург  
2022 г.

**Задание 1.**

В прямом порядке обхода бинарного дерева была получена последовательность узлов FCAEDKHGBJ, а в симметричном - ACDKEFGBHJ. Какая высота дерева?

Полученное дерево:



Ответ: 4

**Задание 2.**



Решение задачи:

Рассмотрим первый класс, где граф представлен матрицей смежности:

1. Указатель m занимает 8 байт => 8\*n – это кол-во байт на указатели.
2. Поскольку нам дана треугольная динамическая матрица, то сумма всех элементов арифметической прогрессии будет равна сумме байт.
3. Составляем уравнение для 1 класса: 8 + 8\*n + (n-1)\*n \* 0.5

Рассмотрим второй класс, который представлен списком смежности:

1. Указатель g занимает 8 байт => 8\*n байт на указатели.
2. Каждый узел списка равен 12 байт, где 4 байта – это int, а ещё 8 – указатель на следующий элемент
3. Известно, что максимальная степень вершин равна 6 => для каждой вершины может быть 6 узлов
4. Составим уравнение: 8+8\*n+12\*6\*n

Составим неравенство из двух уравнений:

8 + 8\*n + 12\*6\*n < 8 + 8\*n + (n-1)\*n\*0.5

8 + 80\*n < 8 + 8\*n + 0.5\*n^2 - 0.5\*n

0.5\*n^2 - 72.5\*n > 0

n\*(n-145) > 0 => n принадлежит (-oo; 0) v (145; +oo) => известно, что n > 0, из этого следует, что n равно наименьшему целому положительному числу, принадлежащему промежутку => n = 146

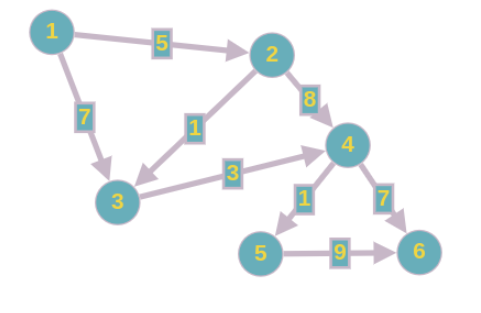
Ответ: 146

**Задание 3.** Написать программу для решения задачи.

Уровень сложности – **повышенный**. Граф представляется двумя способами (матрицей смежности или весов и списками смежности), для тестирования программы требуется создать файлы с описанием графа одним способом (только матрицей или только списками), при считывании данных из файла заполнять параллельно обе структуры хранения (и реализующую представление матрицы смежности (весов), и реализующую представление списков смежных вершин). При выборе структур хранения руководствоваться требованием разумной экономии памяти.

Купцы одного сказочного мира учредили свой ежегодный съезд – Континентальный Купеческий Конгресс, где обсуждают свои проблемы. Каждая страна отправляет на съезд по одному делегату, а расходы на пересечение границ оплачиваются из общей кассы. Посчитайте эти расходы, если известна страна проведения конгресса, купцы следуют на съезд маршрутами, на которых количество пересечений границ минимально, но на каждой границе установлен свой размер пошлины.

Тестовый граф №1:



Матрица смежности представлена в файле matrix.txt в виде:

1. 0
2. 6 6
3. 0 5 7 0 0 0
4. 0 0 1 8 0 0
5. 0 0 0 3 0 0
6. 0 0 0 0 1 7
7. 0 0 0 0 0 9
8. 0 0 0 0 0 0

Число в первой строке – тип представления графа, где 0 – матрица смежности, 1 – список смежности. Вторая строка – количество вершин и конечная вершина, до которой нужно добраться.

Список смежности графа представлен в файле matrix.txt:

1) 1

2) 6 6

3) 2 2 5 3 7

4) 2 3 1 4 8

5) 1 4 3

6) 2 5 1 6 7

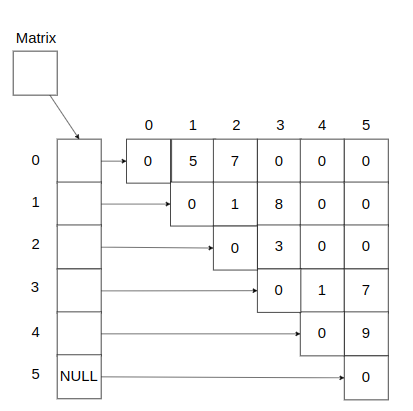
7) 1 6 9

8) 0

Число в первой строке – тип представления графа, где 0 – матрица смежности, 1 – список смежности. Вторая строка – количество вершин и конечная вершина, до которой нужно добраться. Последующие строки: первое число – количество путей (обозначим кол-во переменной k) из вершины n+2, где (n+2) – это номер строки, n – номер вершины; второе число – номер вершины, куда ведёт вершина n, третье число – вес пути. Второе и третье число повторяется k раз.

Ожидаемый результат 53

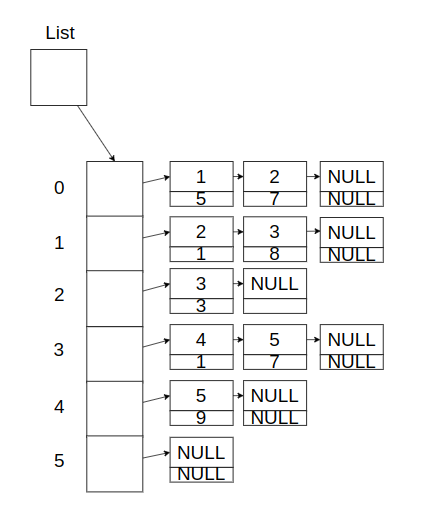
Структура хранения для представления графа матрицей смежности или весов:



Для хранения в виде треугольной матрицы смежности вершин понадобится:

4 – на указатель matrix, 4\*6 = 24 – для массива указателей, где 4 – размер одного указатель,6 – количество вершин => необходимо выделить 4\*6\*(1+5/2) = 72 байт памяти => 4+24+72 = 100 байт памяти

Структура хранения для представления графа списками смежных вершин:

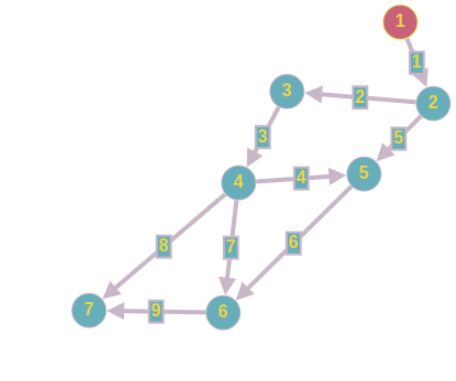


Для хранения в виде треугольной списка смежности вершин понадобится:

4 – на указатель list, 4\*6 = 24– для массива указателей, где 4 – размер одного указатель, 4 – тип данных в матрице (int), 6\*(4+4) = 48 байт – для хранения узлов, где 4 байта на указатель следующего узла и на число, 4\*2\*6 = 48 – для указателей на голову и хвост списка => необходимо выделить 48\*2+24 + 4 =124 байт памяти.

Итог: Матрица смежности выгоднее по памяти.

Тестовый граф №2:



Матрица смежности представлена в файле matrix.txt в виде:

1. 0
2. 6 6
3. 0 5 7 0 0 0
4. 0 0 1 8 0 0
5. 0 0 0 3 0 0
6. 0 0 0 0 1 7
7. 0 0 0 0 0 9
8. 0 0 0 0 0 0

Число в первой строке – тип представления графа, где 0 – матрица смежности, 1 – список смежности. Вторая строка – количество вершин и конечная вершина, до которой нужно добраться.

Список смежности графа представлен в файле matrix.txt:

1) 1

2) 6 6

3) 2 2 5 3 7

4) 2 3 1 4 8

5) 1 4 3

6) 2 5 1 6 7

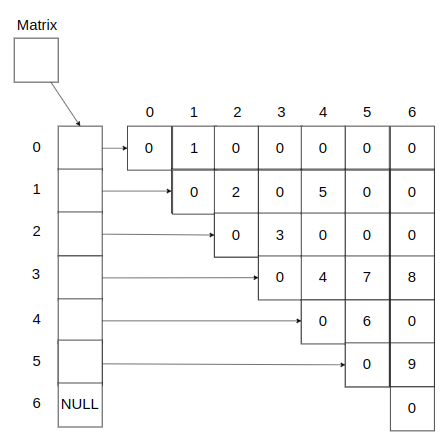
7) 1 6 9

8) 0

Число в первой строке – тип представления графа, где 0 – матрица смежности, 1 – список смежности. Вторая строка – количество вершин и конечная вершина, до которой нужно добраться. Последующие строки: первое число – количество путей (обозначим кол-во переменной k) из вершины n+2, где (n+2) – это номер строки, n – номер вершины; второе число – номер вершины, куда ведёт вершина n, третье число – вес пути. Второе и третье число повторяется k раз.

Ожидаемый результат 70

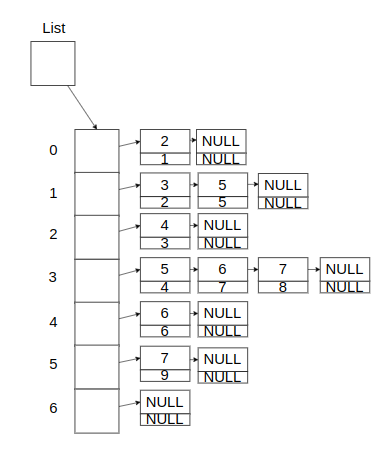
Структура хранения для представления графа матрицей смежности или весов:



Для хранения в виде треугольной матрицы смежности вершин понадобится:

4 – на указатель matrix, 4\*7 = 28– для массива указателей, где 4 – размер одного указатель,6 – количество вершин => необходимо выделить 4\*7\*(1+6/2) = 112 байт памяти => 4+28+112 = 144 байт памяти

Структура хранения для представления графа списками смежных вершин:

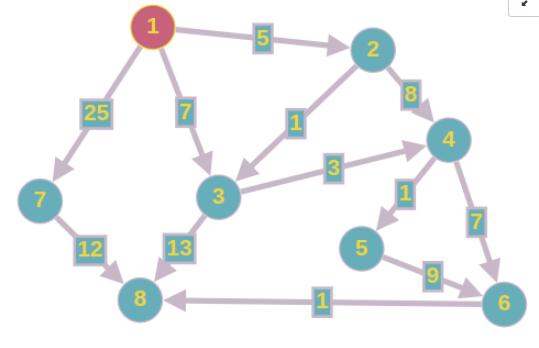


Для хранения в виде треугольной списка смежности вершин понадобится:

4 – на указатель list, 4\*7 = 28– для массива указателей, где 4 – размер одного указатель, 4 – тип данных в матрице (int), 7\*(4+4) = 56 байт – для хранения узлов, где 4 байта на указатель следующего узла и на число, 4\*2\*7 = 56 – для указателей на голову и хвост списка => необходимо выделить 56\*2+28 + 4 =144 байт памяти.

Итог: Матрица смежности и список смежности оба выгодны по памяти.

Тестовый граф №3:



Матрица смежности представлена в файле matrix.txt в виде:

1. 0
2. 8 8
3. 0 5 7 0 0 0 25 0
4. 0 0 1 8 0 0 0 0
5. 0 0 0 3 0 0 0 13
6. 0 0 0 0 1 7 0 0
7. 0 0 0 0 0 9 0 0
8. 0 0 0 0 0 0 0 1
9. 0 0 0 0 0 0 0 12
10. 0 0 0 0 0 0 0 0

Число в первой строке – тип представления графа, где 0 – матрица смежности, 1 – список смежности. Вторая строка – количество вершин и конечная вершина, до которой нужно добраться.

Список смежности графа представлен в файле matrix.txt:

1) 1

2) 6 6

3) 3 2 5 3 7 7 25

4) 2 3 1 4 8

5) 2 4 3 8 13

6) 2 5 1 6 7

7) 1 6 9

8) 1 8 1

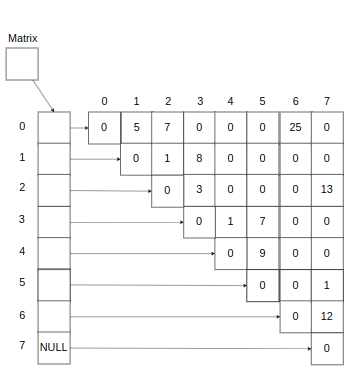
9) 1 8 12

10) 0

Число в первой строке – тип представления графа, где 0 – матрица смежности, 1 – список смежности. Вторая строка – количество вершин и конечная вершина, до которой нужно добраться. Последующие строки: первое число – количество путей (обозначим кол-во переменной k) из вершины n+2, где (n+2) – это номер строки, n – номер вершины; второе число – номер вершины, куда ведёт вершина n, третье число – вес пути. Второе и третье число повторяется k раз.

Ожидаемый результат 71

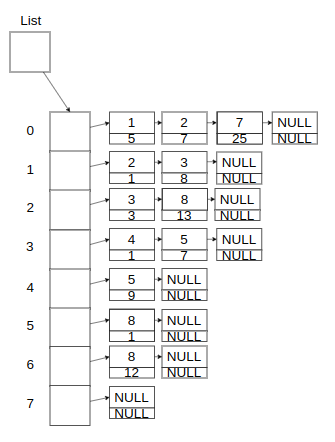
Структура хранения для представления графа матрицей смежности или весов:



Для хранения в виде треугольной матрицы смежности вершин понадобится:

4 – на указатель matrix, 4\*8 = 32– для массива указателей, где 4 – размер одного указатель,8 – количество вершин => необходимо выделить 4\*8\*(1+7/2) = 128 байт памяти => 4+32+128= 164 байт памяти

Структура хранения для представления графа списками смежных вершин:



Для хранения в виде треугольной списка смежности вершин понадобится:

4 – на указатель list, 4\*8 = 32– для массива указателей, где 4 – размер одного указатель, 4 – тип данных в матрице (int), 8\*(4+4) = 64 байт – для хранения узлов, где 4 байта на указатель следующего узла и на число, 4\*2\*8 = 64 – для указателей на голову и хвост списка => необходимо выделить 64\*2+32 + 4 =164 байт памяти.

Итог: Матрица смежности и список смежности оба выгодны по памяти.

Применяемые алгоритмы:

Алгоритм Дейкстры

Текст программы:

Файл «main.cpp»

/\*

Граф представляется двумя способами (матрицей

смежности или весов и списками смежности). Для каждого представления

требуется написать отдельную программу решения задачи, используя

алгоритм, наиболее подходящий для используемой структуры хранения. Для

тестирования программ требуется создать файлы с описанием графа одним

способом (только матрицей или только списками), обе программы должны

уметь заполнять структуры хранения, считывая файлы, как содержащие

матрицы смежности, так и содержащие списки смежных вершин. При выборе

структур хранения руководствоваться требованием разумной экономии

памяти.

Купцы одного сказочного мира учредили свой ежегодный съезд –

Континентальный Купеческий Конгресс, где обсуждают свои проблемы.

Каждая страна отправляет на съезд по одному делегату, а расходы на

пересечение границ оплачиваются из общей кассы. Посчитайте эти расходы,

если известна страна проведения конгресса, купцы следуют на съезд

маршрутами, на которых количество пересечений границ минимально, но на

каждой границе установлен свой размер пошлины.

Файл «main.cpp»

#include <iostream>

#include "Graph.h"

int main() {

Graph graphMatrix;

Graph<ListGraphs> graphList;

try {

// Graph Matrix

graphMatrix.ReadFile("/home/anton/CLionProjects/AlgorithmDijkstra/matrix.txt");

graphMatrix.Log("Print Matrix");

graphMatrix.PrintGraph();

graphMatrix.Log("Answer Matrix = " + std::to\_string(graphMatrix.PriceDijkstra()));

graphList.ReadFile("/home/anton/CLionProjects/AlgorithmDijkstra/matrix.txt");

graphList.Log("Answer List = " + std::to\_string(graphList.PriceDijkstra()));

}

catch(std::string& ex) {

std::cout << ex << '\n';

}

return 0;

}

Файл «Container.h»

#pragma once

#include <iostream>

#include <string>

class Container {

public:

explicit Container() = default;

virtual ~Container() {};

virtual void PrintGraph() = 0;

virtual void ReadFile(std::string filename) = 0;

virtual bool AlgorithmGraphs(const unsigned short startPosition) = 0;

virtual uint32\_t Dijkstra(int src) = 0;

virtual uint32\_t PriceDijkstra() = 0;

public:

template<typename T> void Log(T userText) {

std::cout << "LOG: " << userText << '\n';

}

protected:

unsigned short finalPath, numVer;

unsigned int expenses;

};

Файл «Graph.h»

#pragma once

#include "MatrixGraphs.h"

#include "ListGraphs.h"

template <class Container = MatrixGraphs> class Graph {

public:

Graph() {

container = new Container;

};

void PrintGraph() {

container->PrintGraph();

}

void ReadFile(std::string filename) {

container->ReadFile(filename);

}

uint32\_t Dijkstra(int src) {

return container->Dijkstra(src);

}

uint32\_t PriceDijkstra() {

return container->PriceDijkstra();

}

bool AlgorithmGraphs(const unsigned short startPosition) {

return container->AlgorithmGraphs(startPosition);

}

template<typename T> void Log(T userText) {

container->Log(userText);

}

~Graph() {

delete container;

}

private:

Container\* container;

};

Файл «ListGraphs.cpp»

#include "ListGraphs.h"

#include "fstream"

#include "climits"

ListGraphs::Node\* ListGraphs::CreateNode(int v, int p) {

Node\* newNode = new Node;

newNode->vertex = v;

newNode->price = p;

newNode->next = NULL;

return newNode;

}

void ListGraphs::AddEdge(int src, int dest, int price) {

Node\* newNode = CreateNode(dest, price);

newNode->next = adjLists->adjList[src];

adjLists->adjList[src] = newNode;

newNode = CreateNode(src, price);

newNode->next = adjLists->adjList[dest];

adjLists->adjList[dest] = newNode;

}

ListGraphs::ListGraphs() {

finalPath = numVer = expenses = 0;

}

ListGraphs::~ListGraphs() {

if (adjLists!=NULL) {

for (int i = 0; i<numVer; ++i) {

if (adjLists->adjList[i]!=NULL) {

delete[] adjLists->adjList[i];

}

}

delete[] adjLists;

}

}

void ListGraphs::PrintGraph() {

for (int v = 0; v < numVer; v++)

{

Node\* temp = adjLists->adjList[v];

std::cout << "Vertix: " << v+1 << '\n';

while (temp)

{

std::cout << temp->vertex << " " << temp->price << " ";

temp = temp->next;

}

std::cout << '\n';

}

}

void ListGraphs::ReadFile(std::string filename) {

bool typeMatrix = false;

int tempVertexFrom;

std::ifstream file;

file.open(filename.c\_str());

if (file.peek() == EOF) {

file.close();

throw std::string("File is empty");

}

file >> typeMatrix;

file >> numVer;

file >> finalPath;

adjLists = new Graph;

adjLists->numVert = numVer;

adjLists->adjList = new Node\*[numVer];

for (int i = 0; i < numVer; ++i) {

adjLists->adjList[i] = NULL;

}

if (typeMatrix == true) {

for (int i = 0; i < numVer; ++i) {

int tempVertexFrom;

file >> tempVertexFrom;

for (int j = 0; j < tempVertexFrom; ++j) {

int tempVertexSrc, tempPrice;

file >> tempVertexSrc;

file >> tempPrice;

AddEdge(i, tempVertexSrc, tempPrice);

}

}

}

else {

for (int i = 0; i < numVer; ++i) {

adjLists[i].numVert = 0;

for (int j = 0; j < numVer; ++j) {

file >> tempVertexFrom;

if (tempVertexFrom != 0) {

adjLists[i].numVert++;

adjLists[i].adjList[j]->vertex = j;

adjLists[i].adjList[j]->price = tempVertexFrom;

}

}

}

}

file.close();

}

bool ListGraphs::AlgorithmGraphs(const unsigned short startPosition) {

if (adjLists == NULL) {

Log(123);

return false;

}

return true;

}

ListGraphs::MinHeapNode\* ListGraphs::newMinHeapNode(int v, int dist) {

MinHeapNode\* minHeapNode = new MinHeapNode;

minHeapNode->v = v;

minHeapNode->dist = dist;

return minHeapNode;

}

ListGraphs::MinHeap\* ListGraphs::createMinHeap(int capacity) {

MinHeap\* minHeap = new MinHeap;

minHeap->pos = new int[capacity];

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = new MinHeapNode\*[capacity];

return minHeap;

}

void ListGraphs::swapMinHeapNode(MinHeapNode\*\* a, MinHeapNode\*\* b) {

MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

void ListGraphs::minHeapify(MinHeap\* minHeap, int idx) {

int smallest, left, right;

smallest = idx;

left = 2 \* idx + 1;

right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->dist < minHeap->array[smallest]->dist ) {

smallest = left;

}

if (right < minHeap->size && minHeap->array[right]->dist < minHeap->array[smallest]->dist ) {

smallest = right;

}

if (smallest != idx)

{

MinHeapNode \*smallestNode = minHeap->array[smallest];

MinHeapNode \*idxNode = minHeap->array[idx];

minHeap->pos[smallestNode->v] = idx;

minHeap->pos[idxNode->v] = smallest;

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

int ListGraphs::isEmpty(MinHeap\* minHeap) {

return minHeap->size == 0;

}

ListGraphs::MinHeapNode\* ListGraphs::extractMin(MinHeap\* minHeap) {

if (isEmpty(minHeap)) {

return NULL;

}

MinHeapNode\* root = minHeap->array[0];

MinHeapNode\* lastNode = minHeap->array[minHeap->size - 1];

minHeap->array[0] = lastNode;

minHeap->pos[root->v] = (minHeap->size--)-1;

minHeap->pos[lastNode->v] = 0;

minHeapify(minHeap, 0);

return root;

}

void ListGraphs::decreaseKey(MinHeap\* minHeap, int v, int dist) {

int i = minHeap->pos[v];

minHeap->array[i]->dist = dist;

while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist) {

minHeap->pos[minHeap->array[i]->v] = (i-1)/2;

minHeap->pos[minHeap->array[(i-1)/2]->v] = i;

swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

bool ListGraphs::isInMinHeap(MinHeap \*minHeap, int v) {

if (minHeap->pos[v] < minHeap->size) {

return true;

}

return false;

}

uint32\_t ListGraphs::Dijkstra(int src) {

int V = adjLists->numVert;

int dist[V];

MinHeap\* minHeap = createMinHeap(V);

for (int v = 0; v < V; ++v) {

dist[v] = INT\_MAX;

minHeap->array[v] = newMinHeapNode(v,dist[v]);

minHeap->pos[v] = v;

}

minHeap->array[src] = newMinHeapNode(src, dist[src]);

minHeap->pos[src] = src;

dist[src] = 0;

decreaseKey(minHeap, src, dist[src]);

minHeap->size = V;

while (!isEmpty(minHeap)) {

MinHeapNode\* minHeapNode = extractMin(minHeap);

int u = minHeapNode->v;

Node\* pCrawl = adjLists->adjList[u]->next;

while (pCrawl != NULL) {

int v = pCrawl->vertex;

if (isInMinHeap(minHeap, v) && dist[u] != INT\_MAX && pCrawl->price + dist[u] < dist[v]) {

dist[v] = dist[u] + pCrawl->price;

decreaseKey(minHeap, v, dist[v]);

}

pCrawl = pCrawl->next;

}

}

return dist[numVer-2];

}

uint32\_t ListGraphs::PriceDijkstra() {

uint32\_t summa = 0;

for (int i = 1; i <= numVer; ++i) {

summa += Dijkstra(i);

}

return summa\*2;

}

Файл «ListGraphs.h»

#pragma once

#include "Container.h"

class ListGraphs final:

public Container {

// Structure of the adjacency list

private:

struct Node {

int vertex;

int price;

Node\* next;

};

struct Graph {

int numVert;

Node\*\* adjList;

};

private:

struct MinHeapNode

{

int v;

int dist;

};

struct MinHeap

{

int size;

int capacity;

int \*pos;

MinHeapNode \*\*array;

};

public:

explicit ListGraphs();

~ListGraphs();

void PrintGraph();

uint32\_t Dijkstra(int src);

uint32\_t PriceDijkstra();

void ReadFile(std::string filename);

bool AlgorithmGraphs(const unsigned short startPosition);

private:

Node\* CreateNode(int v, int p);

void AddEdge(int src, int dest, int price);

void swapMinHeapNode(MinHeapNode\*\* a, MinHeapNode\*\* b);

void minHeapify(MinHeap\* minHeap, int idx);

void decreaseKey(MinHeap\* minHeap, int v, int dist);

bool isInMinHeap(MinHeap \*minHeap, int v);

int isEmpty(MinHeap\* minHeap);

MinHeapNode\* newMinHeapNode(int v, int dist);

MinHeap\* createMinHeap(int capacity);

MinHeapNode\* extractMin(MinHeap\* minHeap);

private:

Graph\* adjLists;

};

/\*

\* 6 1 6

\* 2 2 5 3 8

\* 2 3 2 4 3

\* 2 4 4 5 4

\* 1 6 12

\* 1 6 13

\* 0

\*/

Файл «MatrixGraphs.h»

#pragma once

#include "Container.h"

class MatrixGraphs final:

public Container{

public:

MatrixGraphs();

~MatrixGraphs();

uint32\_t Dijkstra(int src);

uint32\_t PriceDijkstra();

void PrintGraph();

void ReadFile(std::string filename);

bool AlgorithmGraphs(const unsigned short startPosition);

private:

int MinDist(unsigned int\* dist, bool\*Tset);

private:

unsigned int\*\* adjMatrix;

};

Файл «MatrixGraphs.cpp»

//

// Created by rskullw on 09.05.22.

//

#include "MatrixGraphs.h"

#include "fstream"

#include "string"

#include "climits"

int MatrixGraphs::MinDist(unsigned int\* dist, bool\*ok)

{

int min=INT\_MAX;

int index;

for(int i=0;i<numVer;i++) {

if (ok[i] == false && dist[i] <= min) {

min = dist[i];

index = i;

}

}

return index;

}

MatrixGraphs::MatrixGraphs() {

numVer = finalPath = expenses = 0;

}

MatrixGraphs::~MatrixGraphs() {

if (adjMatrix != nullptr) {

for (int i = 0; i < numVer; ++i) {

delete [] adjMatrix[i];

}

delete [] adjMatrix;

}

}

void MatrixGraphs::ReadFile(std::string filename) {

bool typeMatrix = false;

int temp = 0, tempVert = 0, tempPrice = 0;

std::ifstream file;

file.open(filename.c\_str());

if (file.peek() == EOF) {

file.close();

throw std::string("File is empty");

}

file >> typeMatrix;

file >> numVer;

file >> finalPath;

adjMatrix = new unsigned int\*[numVer];

if (typeMatrix == 1) {

for (int i = 0; i < numVer; ++i) {

adjMatrix[i] = new unsigned int[numVer];

for (int j = 0; j < numVer; ++j) {

adjMatrix[i][j] = 0;

}

}

for (int i = 0; i < numVer; ++i) {

file >> temp;

for (int j = 0; j < temp; ++j) {

file >> tempVert;

file >> tempPrice;

adjMatrix[i][tempVert - 1] = tempPrice;

}

}

}

else {

for (int i = 0; i < numVer; ++i) {

adjMatrix[i] = new unsigned int[numVer];

for (int j = 0; j < numVer; ++j) {

file >> adjMatrix[i][j];

}

}

}

}

uint32\_t MatrixGraphs::Dijkstra(int src) {

auto\* dist = new unsigned[numVer];

auto\* okVert = new bool[numVer];

int j = 0;

for (int i = 0; i<numVer; ++i) {

dist[i] = INT\_MAX;

okVert[i] = false;

}

dist[src-1] = 0;

for(int i = 0; i<numVer; i++) {

j = MinDist(dist, okVert); // vertex not yet included.

okVert[j]=true;// m with minimum distance included in Tset.

for(int k = 0; k<numVer; k++)

{

// Updating the minimum distance for the particular node.

if(!okVert[k] && adjMatrix[j][k] && dist[j]!=INT\_MAX && dist[j]+ adjMatrix[j][k]<dist[k])

dist[k]=dist[j]+adjMatrix[j][k];

}

}

delete [] okVert;

delete [] dist;

return dist[numVer-1];

}

void MatrixGraphs::PrintGraph() {

for (int i = 0; i < numVer; ++i) {

std::cout << '\n';

std::cout << i+1 << "| ";

for (int j = 0; j < numVer; ++j) {

std::cout << adjMatrix[i][j] << " ";

}

}

std::cout << '\n';}

bool MatrixGraphs::AlgorithmGraphs(const unsigned short startPosition) {

std::cout << "sd";

return true;

}

uint32\_t MatrixGraphs::PriceDijkstra() {

uint32\_t summa = 0;

for (int i = 1; i <= numVer; ++i) {

summa += Dijkstra(i);

}

return summa;

}

Результаты работы программы:

*Скриншоты окна программы для каждого тестового набора.*