|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **Министерство науки и высшего образования Российской Федерации**  Калужский филиал  федерального государственного бюджетного  образовательного учреждения высшего образования  ***«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»***  ***(КФ МГТУ им. Н.Э. Баумана)*** |

**ФАКУЛЬТЕТ** ***ИУК «Информатика и управление»***

**КАФЕДРА** \_\_***ИУК4 «Программное обеспечение ЭВМ, информационные технологии»***

**ДОМАШНЯЯ РАБОТА №2**

**«Решение задач оптимизации при принятии решений»**

**ДИСЦИПЛИНА: «Типы и структуры данных»**

|  |  |  |
| --- | --- | --- |
| Выполнил: студент гр. ИУК4-42Б | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ( Карельский М.К. )  (Подпись) |
| Проверил: | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ( Пчелинцева Н.И. )  (Подпись) |
| Дата сдачи (защиты):  Результаты сдачи (защиты): | | |
|  | - Балльная оценка:  - Оценка: | |

Калуга, 2022

**Цель:** формирование практических навыков создания алгоритмов решения оптимизационных задач.

**Задание:**

1. Изучить виды задач оптимизации при принятии решений.
2. Изучить основные алгоритмы для решения данных задач.
3. Реализовать алгоритм согласно варианту

**Вариант 7**

Реализовать алгоритм поиска максимального потока с множеством истоков и множеством стоков на основе алгоритма поиска в ширину.

**Листинг:**

***Graph.h***

#pragma once

#include <vector>

#include <iostream>

#include <iomanip>

#include <fstream>

#include <queue>

#include <string>

namespace DTAS

{

class IncorrectWeightException

{

public:

IncorrectWeightException(std::string error) : \_error(error) {}

std::string GetError() { return \_error; }

private:

std::string \_error{};

};

template <typename T>

class Graph

{

public:

Graph() : \_weights(nullptr), \_size(), \_elements(), \_outputFile("Output.txt") {}

Graph(const Graph& graph) : \_size(graph.\_size), \_elements(graph.\_elements), \_outputFile(graph.\_outputFile)

{

\_weights = new int\*\* [\_size] {};

for (int i{}; i < \_size; ++i)

{

\_weights[i] = new int\* [\_size] {};

for (int j{}; j < \_size; ++j)

if (graph.\_weights[i][j] != nullptr)

\_weights[i][j] = new int(\*graph.\_weights[i][j]);

}

}

Graph(uint16\_t size, std::vector<T> elements) : \_size(size), \_elements(elements), \_outputFile("Output.txt")

{

\_weights = new int\*\* [\_size] {};

for (int i{}; i < \_size; ++i)

\_weights[i] = new int\* [\_size] {};

}

Graph(std::string inputFile) : Graph(inputFile, "Output.txt") {}

Graph(std::string inputFile, std::string outputFile) : \_outputFile(outputFile)

{

std::ifstream file(inputFile);

file >> \_size;

for (int i{}; i < \_size; ++i)

{

T element{};

file >> element;

\_elements.push\_back(element);

}

\_weights = new int\*\* [\_size] {};

for (int i{}; i < \_size; ++i)

{

\_weights[i] = new int\* [\_size] {};

for (int j{}; j < \_size; ++j)

{

std::string weight{};

file >> weight;

if (weight == "-")

\_weights[i][j] = nullptr;

else if (std::find\_if(weight.begin(), weight.end(), [](char c) { return !std::isdigit(c); }) != weight.end())

throw IncorrectWeightException("Incorrect weight");

else

\_weights[i][j] = new int(std::stoi(weight));

}

}

file.close();

}

~Graph()

{

for (int i{}; i < \_size; ++i)

{

for (int j{}; j < \_size; ++j)

if (\_weights[i][j] != nullptr)

delete \_weights[i][j];

delete[] \_weights[i];

}

delete[] \_weights;

}

bool IsEmpty() { return \_weights == nullptr; }

uint16\_t GetSize() { return \_size; }

T GetElement(uint16\_t index) { return \_elements[index]; }

bool CheckConnection(uint16\_t i, uint16\_t j) { return \_weights[i][j] != nullptr; }

bool GetWeight(uint16\_t i, uint16\_t j, int& weight)

{

if (\_weights[i][j] == nullptr)

return false;

weight = \*\_weights[i][j];

return true;

}

void SetWeight(uint16\_t i, uint16\_t j, int\* weight)

{

if (\_weights[i][j] != nullptr)

delete \_weights[i][j];

\_weights[i][j] = weight;

}

void Clear()

{

if (!IsEmpty())

{

for (int i{}; i < \_size; ++i)

{

for (int j{}; j < \_size; ++j)

if (\_weights[i][j] != nullptr)

delete \_weights[i][j];

delete[] \_weights[i];

}

delete[] \_weights;

\_weights = nullptr;

\_elements.clear();

\_size = 0;

}

}

void Print(int width)

{

if (IsEmpty())

std::cout << "Graph is empty\n";

else

{

std::cout << std::setw(width) << " ";

for (int i{}; i < \_size; ++i)

std::cout << std::setw(width) << \_elements[i];

std::cout << "\n";

for (int i{}; i < \_size; ++i)

{

std::cout << std::setw(width) << \_elements[i];

for (int j{}; j < \_size; ++j)

if (\_weights[i][j] != nullptr)

std::cout << std::setw(width) << \*\_weights[i][j];

else

std::cout << std::setw(width) << "-";

std::cout << "\n";

}

}

}

void Add(T element)

{

int\*\*\* temp = new int\*\* [\_size + 1]{};

for (int j{}; j < \_size + 1; ++j)

temp[j] = new int\* [\_size + 1]{};

for (int j{}; j < \_size; ++j)

for (int k{}; k < \_size; ++k)

if (\_weights[j][k] != nullptr)

temp[j][k] = new int(\*\_weights[j][k]);

for (int i{}; i < \_size; ++i)

{

for (int j{}; j < \_size; ++j)

if (\_weights[i][j] != nullptr)

delete \_weights[i][j];

delete[] \_weights[i];

}

delete[] \_weights;

\_weights = temp;

\_elements.push\_back(element);

++\_size;

}

void Connect(T source, T destination, std::string weight)

{

int s;

for (s = 0; s < \_size; ++s)

if (\_elements[s] == source)

break;

int d;

if (source == destination)

d = s;

else

{

for (d = 0; d < \_size; ++d)

if (\_elements[d] == destination)

break;

}

if (s < \_size && d < \_size)

{

if (weight == "-")

{

delete \_weights[s][d];

\_weights[s][d] = nullptr;

}

else if (std::find\_if(weight[0] == '-' ? weight.begin() + 1 : weight.begin(), weight.end(), [](char c) { return !std::isdigit(c); }) != weight.end())

throw IncorrectWeightException("Incorrect weight");

else

{

delete \_weights[s][d];

\_weights[s][d] = new int(std::stoi(weight));

}

}

}

void Remove(T element)

{

int i;

for (i = 0; i < \_size; ++i)

if (\_elements[i] == element)

break;

if (i < \_size)

{

int\*\*\* temp = nullptr;

if (\_size > 1)

{

temp = new int\*\* [\_size - 1]{};

for (int j{}; j < \_size - 1; ++j)

temp[j] = new int\* [\_size - 1]{};

for (int j{}; j < i; ++j)

{

for (int k{}; k < i; ++k)

if (\_weights[j][k] != nullptr)

temp[j][k] = new int(\*\_weights[j][k]);

for (int k = i + 1; k < \_size; ++k)

if (\_weights[j][k] != nullptr)

temp[j][k - 1] = new int(\*\_weights[j][k]);

}

for (int j = i + 1; j < \_size; ++j)

{

for (int k{}; k < i; ++k)

if (\_weights[j][k] != nullptr)

temp[j - 1][k] = new int(\*\_weights[j][k]);

for (int k = i + 1; k < \_size; ++k)

if (\_weights[j][k] != nullptr)

temp[j - 1][k - 1] = new int(\*\_weights[j][k]);

}

}

for (int j{}; j < \_size; ++j)

{

for (int k{}; k < \_size; ++k)

if (\_weights[j][k] != nullptr)

delete \_weights[j][k];

delete[] \_weights[j];

}

delete[] \_weights;

\_weights = temp;

\_elements.erase(\_elements.begin() + i);

--\_size;

}

}

void DFS(T start)

{

if (IsEmpty())

std::cout << "Graph is empty\n";

else

{

int i;

for (i = 0; i < \_size; ++i)

if (\_elements[i] == start)

break;

if (i < \_size)

{

bool\* visited = new bool[\_size] {};

DFS(i, visited);

delete[] visited;

}

}

}

void BFS(T start)

{

if (IsEmpty())

std::cout << "Graph is empty\n";

else

{

int index;

for (index = 0; index < \_size; ++index)

if (\_elements[index] == start)

break;

if (index < \_size)

{

bool\* visited = new bool[\_size] {};

std::queue<int> waiting{};

waiting.push(index);

visited[index] = true;

while (!waiting.empty())

{

index = waiting.front();

waiting.pop();

std::cout << \_elements[index] << " ";

std::vector<int> unsorted{};

for (int i{}; i < \_size; ++i)

if ((\_weights[index][i] != nullptr) && (!visited[i]))

{

unsorted.push\_back(i);

visited[i] = true;

}

std::sort(unsorted.begin(), unsorted.end(),

[&](int a, int b) { return \*\_weights[index][a] < \*\_weights[index][b]; });

for (int i{}; i < unsorted.size(); ++i)

waiting.push(unsorted[i]);

}

delete[] visited;

}

}

}

std::vector<uint16\_t> FindShortestWay(uint16\_t startIndex, uint16\_t endIndex)

{

if (IsEmpty())

return std::vector<uint16\_t>{};

bool\* visited = new bool[\_size] {};

uint16\_t\* parents = new uint16\_t[\_size]{};

std::queue<int> waiting{};

uint16\_t index = startIndex;

waiting.push(index);

visited[index] = true;

bool endIsReached = false;

while (!waiting.empty())

{

index = waiting.front();

waiting.pop();

std::vector<int> unsorted{};

for (int i{}; i < \_size; ++i)

if ((\_weights[index][i] != nullptr) && (!visited[i]))

{

unsorted.push\_back(i);

visited[i] = true;

parents[i] = index;

if (i == endIndex)

{

endIsReached = true;

break;

}

}

if (endIsReached)

break;

std::sort(unsorted.begin(), unsorted.end(),

[&](int a, int b) { return \*\_weights[index][a] > \*\_weights[index][b]; });

for (uint16\_t i{}; i < unsorted.size(); ++i)

waiting.push(unsorted[i]);

}

if (!endIsReached)

return std::vector<uint16\_t>{};

std::vector<uint16\_t> way{};

index = endIndex;

while (index != startIndex)

{

way.push\_back(index);

index = parents[index];

}

way.push\_back(startIndex);

std::reverse(way.begin(), way.end());

delete[] visited;

delete[] parents;

return way;

}

void PrintMaxDistances(int width)

{

std::ofstream file(\_outputFile);

if (IsEmpty())

{

std::cout << "Graph is empty\n";

file << "Graph is empty";

}

else

{

int\*\*\* maxDistances = new int\*\* [\_size] {};

for (int i{}; i < \_size; ++i)

maxDistances[i] = new int\* [\_size] {};

for (int i{}; i < \_size; ++i)

for (int j{}; j < \_size; ++j)

{

std::vector<Pair> passed{};

maxDistances[i][j] = FindMaxDistance(i, j, passed);

}

std::cout << std::setw(width) << " ";

file << std::setw(width) << " ";

for (int i{}; i < \_size; ++i)

{

std::cout << std::setw(width) << \_elements[i];

file << std::setw(width) << \_elements[i];

}

std::cout << "\n";

file << "\n";

for (int i{}; i < \_size; ++i)

{

std::cout << std::setw(width) << \_elements[i];

file << std::setw(width) << \_elements[i];

for (int j{}; j < \_size; ++j)

if (maxDistances[i][j] != nullptr)

{

std::cout << std::setw(width) << \*maxDistances[i][j];

file << std::setw(width) << \*maxDistances[i][j];

}

else

{

std::cout << std::setw(width) << "-";

file << std::setw(width) << "-";

}

std::cout << "\n";

file << "\n";

}

for (int i{}; i < \_size; ++i)

{

for (int j{}; j < \_size; ++j)

if (maxDistances[i][j] != nullptr)

delete maxDistances[i][j];

delete[] maxDistances[i];

}

delete[] maxDistances;

}

file.close();

}

private:

void DFS(int index, bool\* visited)

{

std::cout << \_elements[index] << " ";

visited[index] = true;

for (int i{}; i < \_size; ++i)

if ((\_weights[index][i] != nullptr) && (!visited[i]))

DFS(i, visited);

}

struct Pair

{

int i{};

int j{};

};

int\* FindMaxDistance(int source, int destination, std::vector<Pair> passed)

{

if (source == destination)

return new int(0);

int\* maxDistance = nullptr;

for (int i{}; i < \_size; ++i)

{

Pair pair{ source, i };

if (\_weights[source][i] != nullptr && std::find\_if(passed.begin(), passed.end(),

[&](Pair a) { return a.i == pair.i && a.j == pair.j || a.i == pair.j && a.j == pair.i; }) == passed.end())

{

passed.push\_back(pair);

int\* distance = FindMaxDistance(i, destination, passed);

passed.pop\_back();

if (distance != nullptr)

{

\*distance += \*\_weights[source][i];

if (maxDistance == nullptr)

maxDistance = distance;

else if (\*distance > \*maxDistance)

{

delete maxDistance;

maxDistance = distance;

}

}

}

}

return maxDistance;

}

int\*\*\* \_weights{};

uint16\_t \_size{};

std::vector<T> \_elements{};

std::string \_outputFile{};

};

}

***Main.cpp***

#include "Graph.h"

#include <iostream>

DTAS::Graph<std::string> EdmondsKarp(DTAS::Graph<std::string> network,

std::vector<uint16\_t> s, std::vector<uint16\_t> t);

int main()

{

DTAS::Graph<std::string> graph("Input.txt");

std::cout << "Network:\n";

graph.Print(4);

std::string input{};

std::cout << "\nInput sources ('-' to stop):\n";

std::vector<uint16\_t> s{};

while (true)

{

std::cout << ">>> ";

std::cin >> input;

if (input == "-")

break;

uint16\_t index;

for (index = 0; index < graph.GetSize(); ++index)

if (graph.GetElement(index) == input)

break;

if (index >= graph.GetSize())

continue;

s.push\_back(index);

}

std::cout << "\nInput sinks ('-' to stop):\n";

std::vector<uint16\_t> t{};

while (true)

{

std::cout << ">>> ";

std::cin >> input;

if (input == "-")

break;

uint16\_t index;

for (index = 0; index < graph.GetSize(); ++index)

if (graph.GetElement(index) == input)

break;

if (index >= graph.GetSize())

continue;

t.push\_back(index);

}

DTAS::Graph<std::string> f = EdmondsKarp(graph, s, t);

std::cout << "\nMax flow:\n";

f.Print(4);

return 0;

}

DTAS::Graph<std::string> EdmondsKarp(DTAS::Graph<std::string> network,

std::vector<uint16\_t> s, std::vector<uint16\_t> t)

{

int weight;

std::vector<std::string> elements{};

elements.push\_back("S");

for (uint16\_t i{}; i < network.GetSize(); ++i)

elements.push\_back(network.GetElement(i));

elements.push\_back("T");

DTAS::Graph<std::string> G(network.GetSize() + 2, elements);

for (uint16\_t i{}; i < network.GetSize(); ++i)

for (uint16\_t j{}; j < network.GetSize(); ++j)

if (network.GetWeight(i, j, weight))

G.SetWeight(i + 1, j + 1, new int(weight));

else

G.SetWeight(i + 1, j + 1, nullptr);

for (uint16\_t i{}; i < s.size(); ++i)

G.SetWeight(0, s[i] + 1, new int(INT\_MAX));

for (uint16\_t i{}; i < t.size(); ++i)

G.SetWeight(t[i] + 1, G.GetSize() - 1, new int(INT\_MAX));

DTAS::Graph<std::string> f(G);

for (uint16\_t i{}; i < G.GetSize(); ++i)

for (uint16\_t j{}; j < G.GetSize(); ++j)

{

if (G.CheckConnection(i, j))

{

f.SetWeight(i, j, new int(0));

f.SetWeight(j, i, new int(0));

}

}

std::vector<uint16\_t> p = G.FindShortestWay(0, G.GetSize() - 1);

while (!p.empty())

{

int c = INT\_MAX;

for (uint16\_t i{}; i < p.size() - 1; ++i)

{

G.GetWeight(p[i], p[i + 1], weight);

if (c > weight)

c = weight;

}

for (uint16\_t i{}; i < p.size() - 1; ++i)

{

f.GetWeight(p[i], p[i + 1], weight);

f.SetWeight(p[i], p[i + 1], new int(weight + c));

f.SetWeight(p[i + 1], p[i], new int(-weight - c));

}

for (uint16\_t i{}; i < p.size() - 1; ++i)

{

if (!G.GetWeight(p[i], p[i + 1], weight))

weight = 0;

if (weight - c == 0)

G.SetWeight(p[i], p[i + 1], nullptr);

else

G.SetWeight(p[i], p[i + 1], new int(weight - c));

if (!G.GetWeight(p[i + 1], p[i], weight))

weight = 0;

if (weight + c == 0)

G.SetWeight(p[i + 1], p[i], nullptr);

else

G.SetWeight(p[i + 1], p[i], new int(weight + c));

}

p = G.FindShortestWay(0, G.GetSize() - 1);

}

DTAS::Graph<std::string> r(network);

for (uint16\_t i{}; i < network.GetSize(); ++i)

for (uint16\_t j{}; j < network.GetSize(); ++j)

if (f.GetWeight(i + 1, j + 1, weight))

r.SetWeight(i, j, new int(weight));

else

r.SetWeight(i, j, nullptr);

return r;

}

**Результат:**

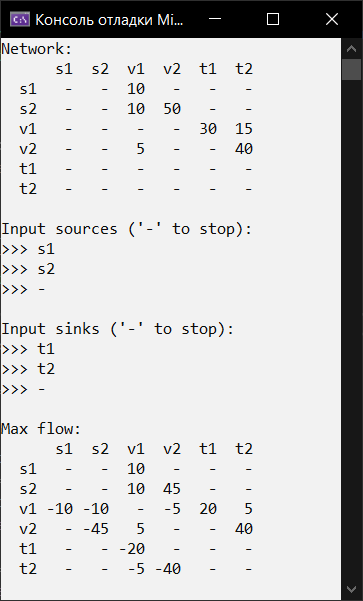


Рис. 1. Результат

**Вывод:** в ходе выполнения домашней работы были получены навыки реализации алгоритма поиска максимального потока с множеством истоков и стоков в решении оптимизационной задачи.

**ОСНОВНАЯ ЛИТЕРАТУРА**

1. Алексеев В.Е. Графы и алгоритмы. Структуры данных. Модели вычислений [Электронный ресурс]/ В.Е. Алексеев, В.А. Таланов. — Электрон. текстовые данные. — М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 153 c. — Режим доступа: <http://www.iprbookshop.ru/52186.html>
2. Вирт Никлаус. Алгоритмы и структуры данных [Электронный ресурс]/ Никлаус Вирт — Электрон. текстовые данные. — Саратов: Профобразование, 2017. — 272 c. — Режим доступа: <http://www.iprbookshop.ru/63821.html>
3. Самуйлов С.В. Алгоритмы и структуры обработки данных [Электронный ресурс]: учебное пособие/ С.В. Самуйлов. — Электрон. текстовые данные. — Саратов: Вузовское образование, 2016. — 132 c. — Режим доступа: <http://www.iprbookshop.ru/47275.html>

**ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА**

1. Костюкова Н.И. Графы и их применение [Электронный ресурс]/ Н.И. Костюкова. — Электрон. текстовые данные. — М.: Интернет Университет Информационных Технологий (ИНТУИТ), 2016. — 147 c. — Режим доступа: <http://www.iprbookshop.ru/52185.html>
2. Сундукова Т.О. Структуры и алгоритмы компьютерной обработки данных [Электронный ресурс]/ Т.О. Сундукова, Г.В. Ваныкина. — Электрон. текстовые данные. — М.: Интернет Университет Информационных Технологий (ИНТУИТ), 2016. — 749 c.— Режим доступа: <http://www.iprbookshop.ru/57384.html>

**Электронные ресурсы:**

1. Научная электронная библиотека <http://elibrary.ru>
2. Электронно-библиотечная система «ЛАНЬ» <http://e.lanbook.com>
3. Электронно-библиотечная система «IPRbooks» <http://www.iprbookshop.ru>
4. Электронно-библиотечная система «Юрайт» <http://www.biblio-online.ru>
5. Электронно-библиотечная система «Университетская библиотека ONLINE» <http://www.biblioclub.ru>