**Міністерство Освіти І НАУКИ України**

**Національний університет "Львівська політехніка"**

Інститут **КНІТ**

Кафедра **ПЗ**

### ЗВІТ

До лабораторної роботи № 7

**З дисципліни:** *“Дослідження операцій”*

**На тему:** *“Розв’язування задачі про багатополюсний максимальний потік*

*за допомогою алгоритму Гоморі-Ху”*

**Лектор:**

доц. каф. ПЗ

Журавчак Л. М.

**Виконав:**

ст. гр. ПЗ-23

Михалевич П.-І.В.

**Прийняла:**

асист. каф. ПЗ

Івасько Н. М.

« \_\_\_\_ » \_\_\_\_\_\_\_\_ 2022 р.

∑= \_\_\_\_\_ .

Львів – 2022

**Тема роботи:** Розв’язування задачі про багатополюсний максимальний потік

за допомогою алгоритму Гоморі-Ху.

**Мета роботи:** Ознайомитись на практиці із алгоритмом Гоморі-Ху

розв’язування задач про максимальний потік.

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

3. Сформуйте задачу про максимальний потік.

Нехай G = (N, А) – орієнтована мережа з одним джерелом s і одним витоком t, і нехай дуги (i, j)А мають обмежену пропускну здатність. Задача про максимальний потік полягає в пошуку таких потоків по дугах, що належать множині А, щоб результативний потік, який витікає із s в t, був максимальним.

6. Що таке пропускна здатність розрізу?

Пропускна здатність розрізу – мінімальна з сум всіх зважених дуг однієї з розділених вершин.

9. На якій ідеї ґрунтується розв’язок задачі про багатополюсний максимальний потік?

Розв’язок задачі про багатополюсний МП ґрунтується на ідеї побудови

на основі первісної мережі дерева-основи або дерева розрізів.

**ЗАВДАННЯ**

1. Отримати індивідуальний варіант завдання.

2. Знайти максимальну інтенсивність інформаційних потоків між всіма вузлами мережі за допомогою алгоритму Гоморі-Ху, якщо задані інтенсивності потоків інформації між безпосередньо пов’язаними вузлами цієї мережі з Додатку № 1 до лабораторної роботи № 7. Побудувати дерево розрізів (вручну).

3. Написати програму розв’язування задачі про максимальний потік за методом Гоморі-Ху згідно з Додатком № 1 до лабораторної роботи № 7.

4. Оформити звіт про виконану роботу.

5. Продемонструвати викладачеві результати, відповісти на запитання стосовно виконання роботи.

**ХІД ВИКОНАННЯ**

1. Індивідуальне завдання.

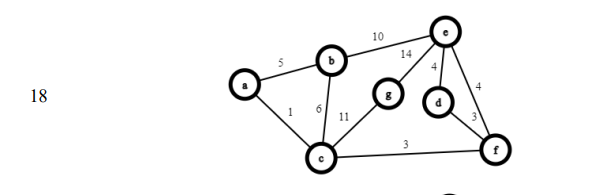


Рис. 1. Індивідуальне завдання

1. Код програми:

#include "HomoriHu.h"

#include <json.h>

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <fstream>

#include <sstream>

#include <limits>

void HomoriHu::Start() {

CreateProblem(false, "lab7\_homori\_main.json");

BuildRoute();

ShowTable();

}

void HomoriHu::CreateProblem(const bool fromConsole, std::string fileName) {

if (fromConsole) {

std::cout << "Enter number of peeks:";

int peeksNumber;

std::cin >> peeksNumber;

m\_costs.resize(peeksNumber, std::vector<double>(peeksNumber, 0));

std::cout << "Enter costs:";

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

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

std::cout << "c[" << i + 1 << "][" << j + 1 << "] = ";

std::cin >> m\_costs[i][j];

}

}

std::cout << "Enter the start peek:";

std::cin >> m\_startPeek;

std::cout << "Enter the end peek:";

std::cin >> m\_endPeek;

}

else {

std::ifstream jsonFile(fileName);

if (jsonFile.fail())

{

throw std::runtime\_error("File was not found.\n");

}

nlohmann::json parsedJson = nlohmann::json::parse(jsonFile);

const auto& parsedJsonCosts = parsedJson.at("Costs");

const auto peeksCount = parsedJson.at("PeeksCount").get<int>();

m\_costs.resize(peeksCount, std::vector<double>(peeksCount, 0));

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

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

m\_costs[i][j] = parsedJsonCosts[i][j].get<double>();

}

}

m\_startPeek = parsedJson.at("StartPeek").get<int>();

m\_endPeek = parsedJson.at("EndPeek").get<int>();

}

}

void HomoriHu::BuildRoute() {

const int vertexCount = m\_costs.size();

m\_step = 1;

std::vector<std::vector<int>> vertexGroups;

std::vector<int> startGroup;

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

startGroup.push\_back(i);

}

vertexGroups.push\_back(startGroup);

auto experienceVertex = -1;

auto prevGroupIndex = -1;

while (m\_step < vertexCount) {

int selectedGroupIndex = -1;

for (int groupIndex = 0; groupIndex < vertexGroups.size(); ++groupIndex) {

if (vertexGroups[groupIndex].size() == 1) {

continue;

}

selectedGroupIndex = groupIndex;

break;

}

if (prevGroupIndex != selectedGroupIndex) {

experienceVertex = FindVertexWithMaxFlow(vertexGroups[selectedGroupIndex]);

}

auto selectedVertex = -1;

for (int i = 0; i < vertexGroups[selectedGroupIndex].size(); ++i) {

if (vertexGroups[selectedGroupIndex][i] == experienceVertex) {

continue;

}

selectedVertex = vertexGroups[selectedGroupIndex][i];

std::vector<int> new\_group(1, experienceVertex);

if (HasConnection(new\_group, vertexGroups[selectedGroupIndex][i])) {

selectedVertex = vertexGroups[selectedGroupIndex][i];

break;

}

}

double min = std::numeric\_limits<double>::max();

int groupIdx = -1;

for (auto connectionIndex = 0; connectionIndex < m\_connections.size(); connectionIndex++) {

if (m\_connections[connectionIndex].GroupIndex1 != selectedGroupIndex && m\_connections[connectionIndex].GroupIndex2 != selectedGroupIndex) {

continue;

}

auto connectedGroupIndex = m\_connections[connectionIndex].GroupIndex1 == selectedGroupIndex ? m\_connections[connectionIndex].GroupIndex2 : m\_connections[connectionIndex].GroupIndex1;

auto new\_group = FindConectedGroupsToSelected(vertexGroups, connectedGroupIndex, selectedGroupIndex);

if (!HasConnection(new\_group, selectedVertex)) {

continue;

}

new\_group.push\_back(selectedVertex);

auto value = FindFlowSum(new\_group);

if (value < min) {

min = value;

groupIdx = connectedGroupIndex;

}

}

std::vector<int> calculationgroup(1, selectedVertex);

auto bestgroup = calculationgroup;

FindInsideSelectedGroup(vertexGroups, selectedGroupIndex, experienceVertex, calculationgroup, bestgroup, min, groupIdx);

std::vector<int> new\_group(1, selectedVertex);

auto value = FindFlowSum(new\_group);

if (value < min) {

vertexGroups.push\_back(new\_group);

vertexGroups[selectedGroupIndex].erase(std::find(vertexGroups[selectedGroupIndex].begin(), vertexGroups[selectedGroupIndex].end(), selectedVertex));

m\_connections.push\_back(Connection{ selectedGroupIndex, -1, int(vertexGroups.size() - 1), selectedVertex, value });

}

else {

if (groupIdx == -1) {

vertexGroups.push\_back(bestgroup);

for (const auto& el : bestgroup) {

vertexGroups[selectedGroupIndex].erase(std::find(vertexGroups[selectedGroupIndex].begin(), vertexGroups[selectedGroupIndex].end(), el));

}

m\_connections.push\_back(Connection{ selectedGroupIndex, -1, int(vertexGroups.size() - 1), -1, min });

}

else {

vertexGroups.push\_back(bestgroup);

for (const auto& el : bestgroup) {

vertexGroups[selectedGroupIndex].erase(std::find(vertexGroups[selectedGroupIndex].begin(), vertexGroups[selectedGroupIndex].end(), el));

}

for (auto connectionIndex = 0; connectionIndex < m\_connections.size(); connectionIndex++) {

if ((m\_connections[connectionIndex].GroupIndex1 == selectedGroupIndex && m\_connections[connectionIndex].GroupIndex2 == groupIdx)

|| (m\_connections[connectionIndex].GroupIndex1 == groupIdx && m\_connections[connectionIndex].GroupIndex2 == selectedGroupIndex)) {

if (m\_connections[connectionIndex].GroupIndex1 == groupIdx) {

m\_connections[connectionIndex].GroupIndex2 = int(vertexGroups.size() - 1);

}

else {

m\_connections[connectionIndex].GroupIndex1 = int(vertexGroups.size() - 1);

}

break;

}

}

m\_connections.push\_back(Connection{ selectedGroupIndex, -1, int(vertexGroups.size() - 1), -1, min });

}

}

prevGroupIndex = selectedGroupIndex;

m\_step++;

}

for (auto connectionIndex = 0; connectionIndex < m\_connections.size(); connectionIndex++){

m\_connections[connectionIndex].VertexIndex1 = vertexGroups[m\_connections[connectionIndex].GroupIndex1][0];

m\_connections[connectionIndex].VertexIndex2 = vertexGroups[m\_connections[connectionIndex].GroupIndex2][0];

}

}

bool HomoriHu::HasConnection(const std::vector<int>& group, int vertex) {

for (int i = 0; i < group.size(); ++i) {

if (m\_costs[group[i]][vertex] != -1) {

return true;

}

}

return false;

}

bool HomoriHu::HasConnection(const std::vector<int>& group1, const std::vector<int>& group2) {

for (int i = 0; i < group1.size(); ++i) {

for (int j = 0; j < group2.size(); ++j) {

if (m\_costs[group1[i]][group2[j]] != -1) {

return true;

}

}

}

return false;

}

int HomoriHu::FindVertexWithMaxFlow(std::vector<int> selectedGroup) {

const int vertexCount = m\_costs.size();

std::vector<std::pair<int, double>> maxValues(vertexCount, {-1, 0});

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

if (std::find(selectedGroup.begin(), selectedGroup.end(), i) == selectedGroup.end()) {

continue;

}

maxValues[i].first = i;

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

if (m\_costs[i][j] == -1) {

continue;

}

maxValues[i].second += m\_costs[i][j];

}

}

return (\*std::max\_element(maxValues.begin(), maxValues.end(), [](const auto& el1, const auto& el2) { return el1.second < el2.second; })).first;

}

void HomoriHu::FindInsideSelectedGroup(

const std::vector<std::vector<int>>& vertexGroups,

const int selectedGroupIndex,

const int expensiveVertex,

const std::vector<int>& calculationgroup,

std::vector<int>& bestgroup,

double& min,

int& groupIdx) {

for (auto selectedGroupVertexIndex = 0; selectedGroupVertexIndex < vertexGroups[selectedGroupIndex].size(); ++selectedGroupVertexIndex) {

auto new\_calculation\_group = calculationgroup;

if (vertexGroups[selectedGroupIndex][selectedGroupVertexIndex] == expensiveVertex) {

continue;

}

if (std::find(calculationgroup.begin(), calculationgroup.end(), vertexGroups[selectedGroupIndex][selectedGroupVertexIndex]) != calculationgroup.end()) {

continue;

}

if (!HasConnection(calculationgroup, vertexGroups[selectedGroupIndex][selectedGroupVertexIndex])) {

continue;

}

new\_calculation\_group.push\_back(vertexGroups[selectedGroupIndex][selectedGroupVertexIndex]);

auto value = FindFlowSum(new\_calculation\_group);

if (value < min) {

min = value;

groupIdx = -1;

bestgroup = new\_calculation\_group;

}

for (auto connectionIndex = 0; connectionIndex < m\_connections.size(); connectionIndex++) {

if (m\_connections[connectionIndex].GroupIndex1 != selectedGroupIndex || m\_connections[connectionIndex].GroupIndex2 != selectedGroupIndex) {

continue;

}

auto connectedGroupIndex = m\_connections[connectionIndex].GroupIndex1 == selectedGroupIndex ? m\_connections[connectionIndex].GroupIndex2 : m\_connections[connectionIndex].GroupIndex1;

auto new\_group = FindConectedGroupsToSelected(vertexGroups, connectedGroupIndex, selectedGroupIndex);

if (!HasConnection(new\_group, new\_calculation\_group)) {

continue;

}

for (int j = 0; j < new\_calculation\_group.size(); ++j) {

new\_group.push\_back(new\_calculation\_group[j]);

}

auto value = FindFlowSum(new\_group);

if (value < min) {

min = value;

groupIdx = connectedGroupIndex;

bestgroup = new\_calculation\_group;

}

}

FindInsideSelectedGroup(vertexGroups, selectedGroupIndex, expensiveVertex, new\_calculation\_group, bestgroup, min, groupIdx);

}

}

std::vector<int> HomoriHu::FindConectedGroupsToSelected(const std::vector<std::vector<int>>& vertexGroups, const int selectedGroupIndex, const int previousGroupIndex) {

int count = 0;

std::vector<int> result\_vertices;

for (auto connectionIndex = 0; connectionIndex < m\_connections.size(); connectionIndex++) {

if (m\_connections[connectionIndex].GroupIndex1 != selectedGroupIndex && m\_connections[connectionIndex].GroupIndex2 != selectedGroupIndex) {

if ((connectionIndex == m\_connections.size() - 1) && count == 0) {

return vertexGroups[selectedGroupIndex];

}

continue;

}

if ((m\_connections[connectionIndex].GroupIndex1 == selectedGroupIndex && m\_connections[connectionIndex].GroupIndex2 == previousGroupIndex)

|| (m\_connections[connectionIndex].GroupIndex1 == previousGroupIndex && m\_connections[connectionIndex].GroupIndex2 == selectedGroupIndex)) {

if ((connectionIndex == m\_connections.size() - 1) && count == 0) {

return vertexGroups[selectedGroupIndex];

}

continue;

}

++count;

auto nextGroup = m\_connections[connectionIndex].GroupIndex1 == selectedGroupIndex ? m\_connections[connectionIndex].GroupIndex2 : m\_connections[connectionIndex].GroupIndex1;

auto vertices = std::move(FindConectedGroupsToSelected(vertexGroups, nextGroup, selectedGroupIndex));

result\_vertices.insert(result\_vertices.end(), vertices.begin(), vertices.end());

; /\*for (const auto& el : vertices) {

result\_vertices.push\_back(el);

}\*/

}

for (const auto& el : vertexGroups[selectedGroupIndex]) {

result\_vertices.push\_back(el);

}

return result\_vertices;

}

double HomoriHu::FindWay(const int selectedVertexIndex, const int previousVertexIndex, const int neededVertexIndex) {

int count = 0;

double result\_way = -1;

for (auto connectionIndex = 0; connectionIndex < m\_connections.size(); connectionIndex++) {

if (m\_connections[connectionIndex].VertexIndex1 != selectedVertexIndex && m\_connections[connectionIndex].VertexIndex2 != selectedVertexIndex) {

if ((connectionIndex == m\_connections.size() - 1) && count == 0) {

return -1;

}

continue;

}

if ((m\_connections[connectionIndex].VertexIndex1 == selectedVertexIndex && m\_connections[connectionIndex].VertexIndex2 == previousVertexIndex)

|| (m\_connections[connectionIndex].VertexIndex1 == previousVertexIndex && m\_connections[connectionIndex].VertexIndex2 == selectedVertexIndex)) {

if ((connectionIndex == m\_connections.size() - 1) && count == 0) {

return -1;

}

continue;

}

if (m\_connections[connectionIndex].VertexIndex1 == neededVertexIndex || m\_connections[connectionIndex].VertexIndex2 == neededVertexIndex) {

return m\_connections[connectionIndex].Value;

}

++count;

auto nextGroup = m\_connections[connectionIndex].VertexIndex1 == selectedVertexIndex ? m\_connections[connectionIndex].VertexIndex2 : m\_connections[connectionIndex].VertexIndex1;

auto value = FindWay(nextGroup, selectedVertexIndex, neededVertexIndex);

if (value == -1) {

continue;

}

result\_way = m\_connections[connectionIndex].Value < value ? m\_connections[connectionIndex].Value : value;

}

return result\_way;

}

double HomoriHu::FindFlowSum(const std::vector<int>& group) {

double result = 0.f;

for (int i = 0; i < group.size(); ++i) {

const int rowIdx = group[i];

for (int j = 0; j < m\_costs.size(); ++j) {

if (std::find(group.begin(), group.end(), j) != group.end() || m\_costs[rowIdx][j] == -1) {

continue;

}

result += m\_costs[rowIdx][j];

}

}

return result;

}

void HomoriHu::ShowTable() {

const auto vertexCount = m\_costs.size();

m\_table.resize(vertexCount, std::vector<double>(vertexCount, -1.f));

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

for (int j = i+1; j < vertexCount; ++j) {

for (auto connectionIndex = 0; connectionIndex < m\_connections.size(); connectionIndex++) {

if (m\_connections[connectionIndex].VertexIndex1 != i && m\_connections[connectionIndex].VertexIndex2 != i) {

continue;

}

if ((m\_connections[connectionIndex].VertexIndex1 == i && m\_connections[connectionIndex].VertexIndex2 == j)

|| (m\_connections[connectionIndex].VertexIndex1 == j && m\_connections[connectionIndex].VertexIndex2 == i)) {

m\_table[i][j] = m\_connections[connectionIndex].Value;

break;

}

auto connectedGroupIndex = m\_connections[connectionIndex].VertexIndex1 == i ? m\_connections[connectionIndex].VertexIndex2 : m\_connections[connectionIndex].VertexIndex1;

auto way = FindWay(connectedGroupIndex, i, j);

if(m\_table[i][j] == -1 && way != -1){

m\_table[i][j] = way < m\_connections[connectionIndex].Value ? way : m\_connections[connectionIndex].Value;

}

}

}

}

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

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

m\_table[i][j] = m\_table[j][i];

}

}

std::cout << "Result table:\n";

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

std::cout << std::setprecision(1) << std::left << std::fixed << std::setw(4) << "[";

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

std::cout << std::setprecision(1) << std::left << std::fixed << std::setw(7) << m\_table[i][j];

}

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

}

}

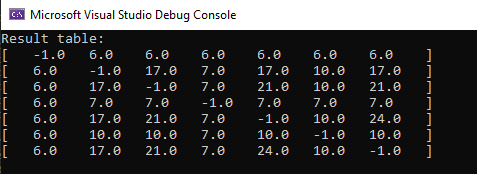


Рис. 2. Результати програми

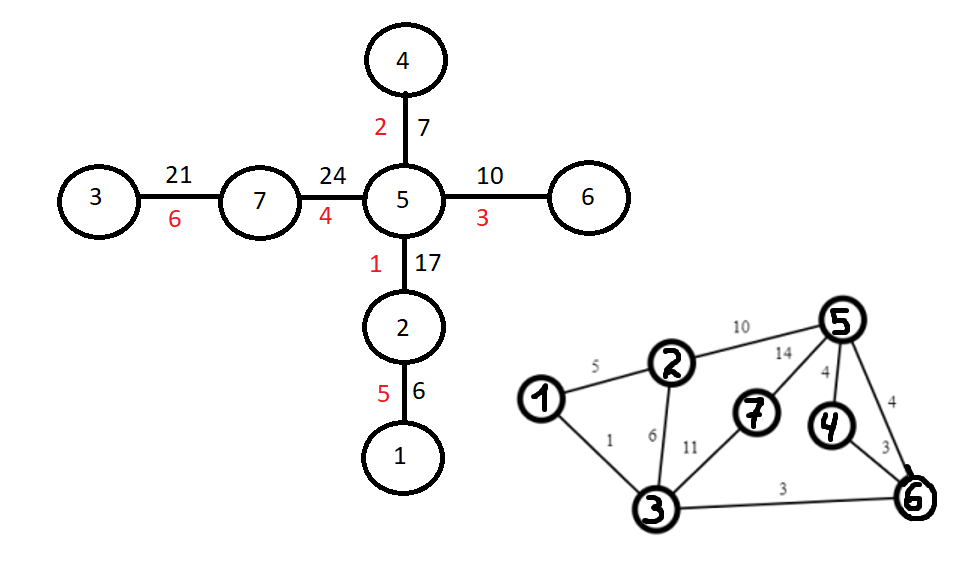


Рис. 3. Дерево вузлів

**ВИСНОВКИ**

На цій лабораторній роботі я ознайомився на практиці із алгоритмом Гоморі-Ху

розв’язування задач про максимальний потік, реалізував та розв’язав за допомогою даного алгоритму мережеву задачу з індивідуального варіанту.