**Міністерство Освіти І НАУКИ України**

**Національний університет "Львівська політехніка"**

Інститут **КНІТ**

Кафедра **ПЗ**

### ЗВІТ

До лабораторної роботи № 8

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

**На тему:** *“* *Ігрові задачі дослідження операцій”*

**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:** Ігрові задачі дослідження операцій.

**Мета роботи:** Ознайомитись на практиці із розв’язуванням матричних ігор з

використанням симплекс-методу.

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

3. Яка гра називається грою з нульовою сумою?

У такій грі сума виграшів всіх гравців в кожній партії рівна нулю, тобто в цій грі загальний капітал гравців не змінюється, а перерозподіляється між ними залежно від результатів гри.

6. Що таке хід гри?

Вибір способу дій гравців на кожному етапі гри та виплату для кожного з гравців після завершення етапу гри.

9. Сформулюйте принцип мінімаксу.

Обирається така стратегія, яка за найгірших умов забезпечить максимальний виграш. При цьому вважається, що суперник буде робити найкращі свої ходи, тобто оптимальним чином (зі своєї точки зору) протидіяти.

**ЗАВДАННЯ**

1. Отримати індивідуальний варіант завдання.

2. Подати гру, задану матрицею згідно з варіантом в Додатку до лабораторної роботи № 8, у вигляді задач ЛП та розв’язати їх.

3. Оформити звіт про роботу, який повинен містити всі ітерації обох

алгоритмів та розрахунки з поясненнями, за допомогою яких отримано результат.

4. Продемонструвати викладачеві результати, відповісти на запитання

стосовно виконання роботи.

**ХІД ВИКОНАННЯ**

1. Індивідуальне завдання.

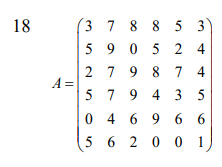


Рис. 1. Індивідуальне завдання

1. Код програми:

void HybridGame::Start() {

CreateProblem(false, "Hybrid\_main.json");

if (FindBoundaries()) {

return;

}

SimplifyProblem();

CreateSymplexTables();

ShowResults();

}

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

if (fromConsole) {

std::cout << "Enter number of A strategy:";

int strategyA;

std::cin >> strategyA;

std::cout << "Enter number of B strategy:";

int strategyB;

std::cin >> strategyB;

m\_table.resize(strategyA, std::vector<int>(strategyB, 0));

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

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

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

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

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

}

}

}

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 parsedJsonStrategyACount = parsedJson.at("StrategyACount").get<int>();

const auto parsedJsonStrategyBCount = parsedJson.at("StrategyBCount").get<int>();

const auto matrix = parsedJson.at("Matrix");

m\_table.resize(parsedJsonStrategyACount, std::vector<int>(parsedJsonStrategyBCount, 0));

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

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

m\_table[i][j] = matrix[i][j].get<int>();

}

}

}

}

bool HybridGame::FindBoundaries() {

std::vector<int> maxInCols = std::vector<int>(m\_table[0].size(), std::numeric\_limits<int>::min());

std::vector<int> minInRows = std::vector<int>(m\_table.size(), std::numeric\_limits<int>::max());

for (int i = 0; i < m\_table.size(); ++i) {

for (int j = 0; j < m\_table[0].size(); ++j) {

if (m\_table[i][j] < minInRows[i]) {

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

}

if (m\_table[i][j] > maxInCols[j]) {

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

}

}

}

auto lowBoundary = \*std::max\_element(minInRows.begin(), minInRows.end());

auto highBoundary = \*std::min\_element(maxInCols.begin(), maxInCols.end());

if (lowBoundary == highBoundary) {

std::cout << "\nProblem is solved with cost = " << highBoundary << "\n";

return true;

}

return false;

}

void HybridGame::SimplifyProblem() {

std::vector<int> removedRows;

std::vector<int> removedCols;

for (int mainRow = 0; mainRow < m\_table.size(); ++mainRow) {

if (std::find(removedRows.begin(), removedRows.end(), mainRow) != removedRows.end()) {

continue;

}

for (int i = 0; i < m\_table.size(); ++i) {

if (i == mainRow) {

continue;

}

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

continue;

}

int count = 0;

for (int j = 0; j < m\_table[0].size(); ++j) {

if (m\_table[i][j] <= m\_table[mainRow][j]) {

++count;

}

}

if (count == m\_table[0].size()) {

removedRows.push\_back(i);

}

else if (count == 0) {

removedRows.push\_back(mainRow);

break;

}

}

}

for (int mainCol = 0; mainCol < m\_table[0].size(); ++mainCol) {

if (std::find(removedCols.begin(), removedCols.end(), mainCol) != removedCols.end()) {

continue;

}

for (int i = 0; i < m\_table[0].size(); ++i) {

if (i == mainCol) {

continue;

}

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

continue;

}

int count = 0;

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

if (m\_table[j][i] <= m\_table[j][mainCol]) {

++count;

}

}

if (count == m\_table.size()) {

removedCols.push\_back(i);

}

else if (count == 0) {

removedCols.push\_back(mainCol);

break;

}

}

}

for (int i = 0; i < m\_table.size(); ++i) {

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

continue;

}

std::vector<int> row;

for (int j = 0; j < m\_table[0].size(); ++j) {

if (std::find(removedCols.begin(), removedCols.end(), j) != removedCols.end()) {

continue;

}

row.push\_back(m\_table[i][j]);

}

m\_simplifiedTable.push\_back(row);

}

}

void HybridGame::CreateSymplexTables() {

m\_syntexSolver.CreateProblem(m\_simplifiedTable);

auto result = m\_syntexSolver.Start();

m\_v = 1 / result[0];

m\_p.reserve(m\_simplifiedTable.size());

m\_q.reserve(m\_simplifiedTable[0].size());

for (int i = 1; i <= m\_simplifiedTable[0].size(); ++i) {

m\_q.push\_back(result[i]);

}

for (int i = m\_simplifiedTable[0].size()+1; i <= m\_simplifiedTable[0].size() + m\_simplifiedTable.size(); ++i) {

m\_p.push\_back(result[i]);

}

}

void HybridGame::ShowResults() {

std::cout << "\n\nResults\n\nTable:\n";

for (int i = 0; i < m\_simplifiedTable.size(); ++i) {

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

for (int j = 0; j < m\_simplifiedTable[0].size(); ++j) {

std::cout << std::setprecision(2) << std::left << std::fixed << std::setw(7) << m\_simplifiedTable[i][j];

}

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

}

std::cout << "\n\nGame Price\n\nv = [1/F] = " << m\_v;

std::cout << "\n\nProbabilities:\n\nFor player A:\n";

for (int i = 0; i < m\_p.size(); ++i) {

std::cout << "A[" << i + 1 << "] = " << m\_p[i] \* m\_v << "\n";

}

std::cout << "\nFor player B:\n";

for (int i = 0; i < m\_q.size(); ++i) {

std::cout << "B[" << i + 1 << "] = " << m\_q[i] \* m\_v << "\n";

}

}

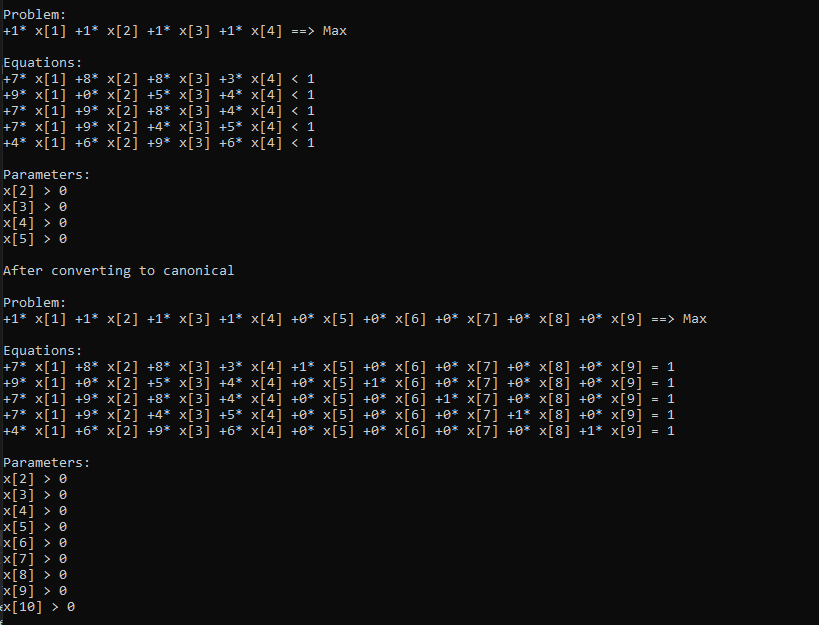


Рис. 2. Результати програми

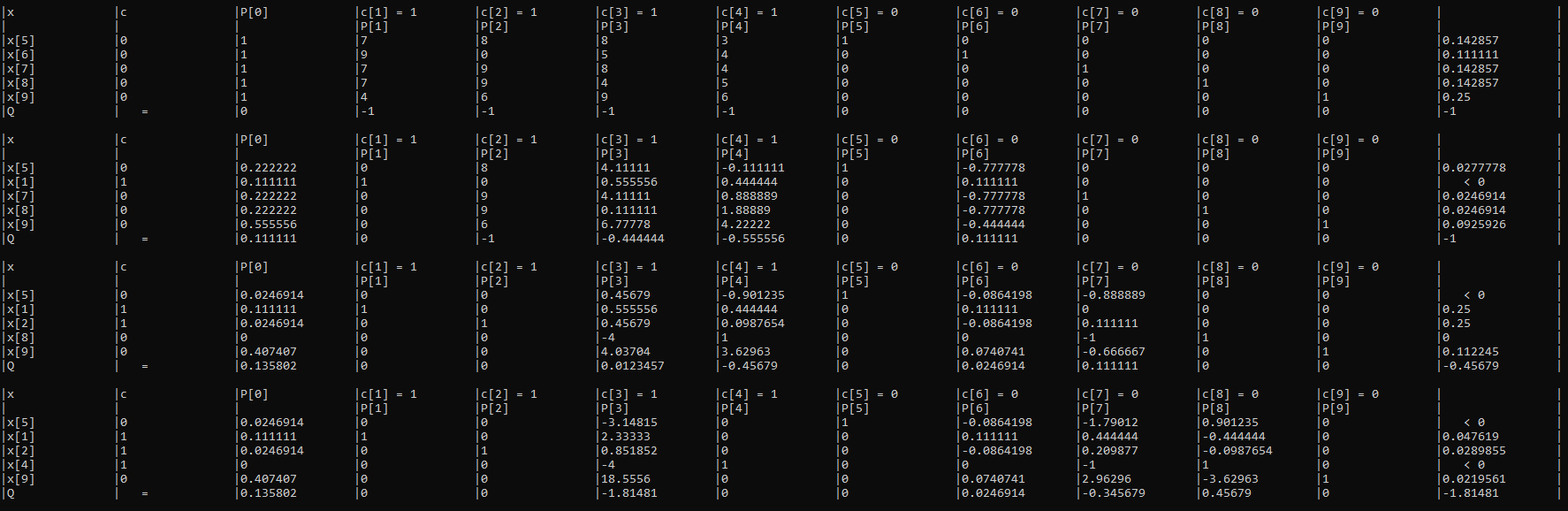


Рис. 3. Результати програми

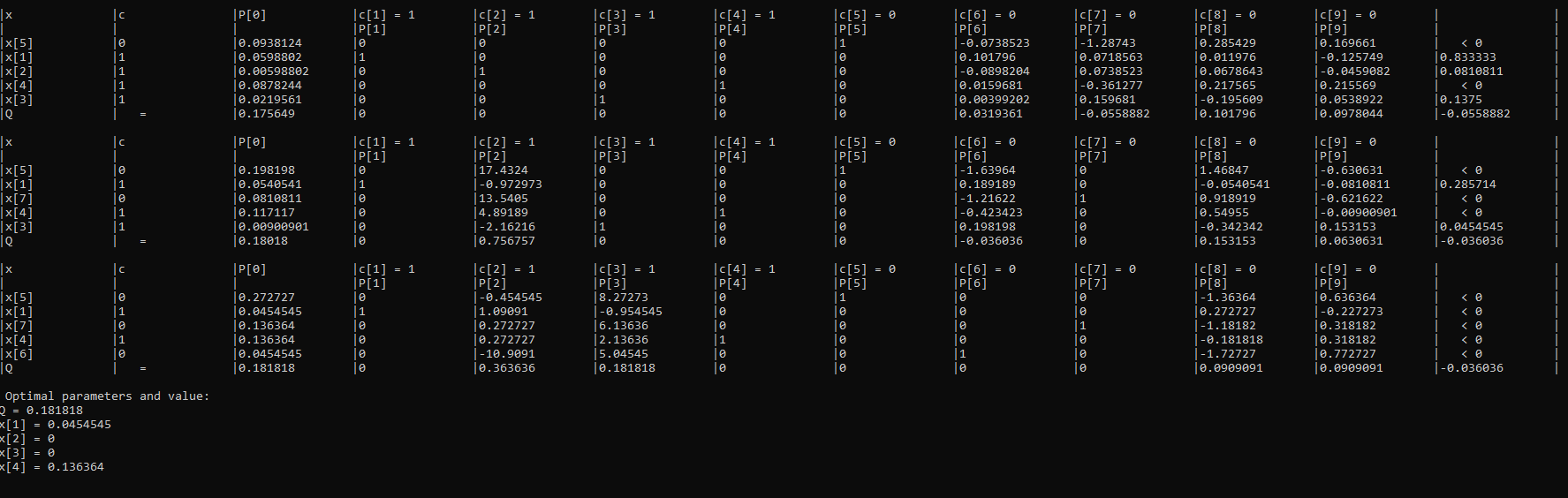


Рис. 4. Результати програми

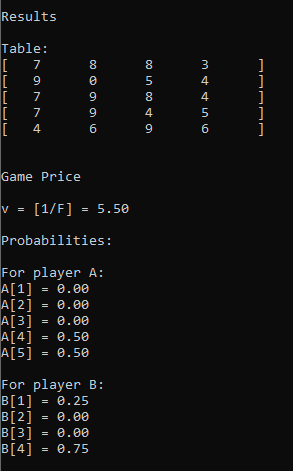


Рис. 5. Результати програми

**ВИСНОВКИ**

На цій лабораторній роботі я ознайомився на практиці із розв’язуванням матричних ігор з використанням симплекс-методу.