МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА» (БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

**Лабораторная работа №4**по дисциплине: «Исследование операций»

Выполнил: ст. группы ПВ-211

Медведев Дмитрий Сергеевич

Проверили:

Куртова Лилиана Николаевна

Вирченко Юрий

Петрович

Белгород 2023 г.

Закрытая транспортная задача

# Вариант 8

**Цель работы:** изучить математическую модель транспортной задачи, овладеть методами решения этой задачи.

**Ход работы**

****

1. Изучить содержательную и математическую постановки закрытой транспортной задачи, методы нахождения первого опорного решения ее системы ограничений. Изучить понятие цикла пересчета в матрице перевозок. Овладеть распределительным методом и методом потенциалов, а также их алгоритмами.
2. Составить и отладить программы решения транспортной задачи распределительным методом и методом потенциалов.

**Написанная библиотека:**

#include **"transport.h"**#include **<utility>**TransportTable::TransportTable(Matrix costMatrix, vector<**float**> stock,  
 vector<**float**> requests) {  
 **this**->costMatrix = costMatrix;  
 **this**->stock = std::move(stock);  
 **this**->requests = std::move(requests);  
  
 planTable = vector<vector<PlanTableElement >>();  
 **for** (**int** row = 0; row < costMatrix.getRows(); row++) {  
 planTable.emplace\_back(costMatrix.getColumns());  
 **for** (**int** column = 0; column < costMatrix.getColumns(); column++) {  
 planTable[row][column].isBasic = **false**;  
 planTable[row][column].value = 0;  
 }  
 }  
}  
  
**void** TransportTable::fillTransportTableMinValue() {  
 **for** (**int** row = 0; row < planTable.size(); row++) {  
 **for** (**int** column = 0; column < planTable[0].size(); column++) {  
 planTable[row][column].isBasic = **true**;  
 }  
 }  
  
 vector<**float**> stockTmp = stock;  
 vector<**float**> requestsTmp = requests;  
  
 **bool** isFull = **false**;  
 **while** (!isFull) {  
 Position minValuePosition = findMinValuePosition();  
  
 **if** (fcmp(requestsTmp[minValuePosition.column],  
 stockTmp[minValuePosition.row])) {  
 planTable[minValuePosition.row][minValuePosition.column].value = requestsTmp[minValuePosition.column];  
 requestsTmp[minValuePosition.column] = 0;  
 stockTmp[minValuePosition.row] = 0;  
  
 **for** (**int** row = 0; row < planTable.size(); row++) {  
 **if** (row != minValuePosition.row && planTable[row][minValuePosition.column].value == 0) {  
 planTable[row][minValuePosition.column].isBasic = **false**;  
 }  
 }  
  
 **for** (**int** column = 0; column < planTable.size(); column++) {  
 **if** (column != minValuePosition.column && planTable[minValuePosition.row][column].value == 0) {  
 planTable[minValuePosition.row][column].isBasic = **false**;  
 }  
 }  
 } **else if** (requestsTmp[minValuePosition.column] < stockTmp[minValuePosition.row]) {  
 planTable[minValuePosition.row][minValuePosition.column].value = requestsTmp[minValuePosition.column];  
 stockTmp[minValuePosition.row] -= requestsTmp[minValuePosition.column];  
 requestsTmp[minValuePosition.column] = 0;  
  
 **for** (**int** row = 0; row < planTable.size(); row++) {  
 **if** (row != minValuePosition.row && planTable[row][minValuePosition.column].value == 0) {  
 planTable[row][minValuePosition.column].isBasic = **false**;  
 }  
 }  
 } **else** {  
 planTable[minValuePosition.row][minValuePosition.column].value = stockTmp[minValuePosition.row];  
 requestsTmp[minValuePosition.column] -= stockTmp[minValuePosition.row];  
 stockTmp[minValuePosition.row] = 0;  
  
 **for** (**int** column = 0; column < planTable.size(); column++) {  
 **if** (column != minValuePosition.column && planTable[minValuePosition.row][column].value  
 == 0) {  
 planTable[minValuePosition.row][column].isBasic = **false**;  
 }  
 }  
 }  
  
 isFull = checkIfTableIsFull();  
 }  
}  
  
**bool** TransportTable::checkIfTableIsFull() {  
 **for** (**int** row = 0; row < planTable.size(); row++) {  
 **for** (**int** column = 0; column < planTable[0].size(); column++) {  
 **if** (planTable[row][column].isBasic && fcmp(planTable[row][column].value, 0))  
 **return false**;  
 }  
 }  
  
 **return true**;  
}  
  
Position TransportTable::findMinValuePosition() {  
 Position minValuePosition(0, 0);  
 **float** minValue = **INT32\_MAX**;  
  
 **for** (**int** row = 0; row < planTable.size(); row++) {  
 **for** (**int** column = 0; column < planTable[0].size(); column++) {  
 **if** (planTable[row][column].isBasic && fcmp(planTable[row][column].value, 0)  
 && costMatrix.getData(row, column) < minValue) {  
 minValuePosition = Position(row, column);  
 minValue = costMatrix.getData(row, column);  
 }  
 }  
 }  
  
 **return** minValuePosition;  
}  
  
**float** TransportTable::countCycleGamma(Sequence cycle) {  
 **float** sum = 0;  
  
 **for** (**int** i = 0; i < cycle.positions.size() - 1; i++) {  
 **float** nextValue = costMatrix.getData(cycle.positions[i].row, cycle.positions[i].column);  
 **if** (i % 2 == 0) {  
 sum += nextValue;  
 } **else** {  
 sum -= nextValue;  
 }  
 }  
  
 **return** sum;  
}  
  
Sequence TransportTable::findCycle(Position start) {  
 Sequence sequence;  
 sequence.positions.push\_back(start);  
  
 **return** \_findCycle(sequence, Direction::***Any***);  
}  
  
Sequence TransportTable::\_findCycle(Sequence sequence, Direction direction) {  
 **if** (sequence.checkIfCycle()) {  
 **return** sequence;  
 } **else** {  
 Position currentPosition = sequence.getPosition(-1);  
  
 **if** (direction == Direction::***Vertical*** || direction == Direction::***Any***) {  
 *//Проход вверх* **while** (currentPosition.row >= 0) {  
 **bool** isCurrentPositionInSequence = sequence.checkIfPositionInSequence(currentPosition);  
  
 **if** (getPlanTableElement(currentPosition).isBasic && !isCurrentPositionInSequence ||  
 sequence.positions.size() > 2 && currentPosition == sequence.positions[0]) {  
 Sequence newSequence = sequence;  
 newSequence.positions.push\_back(currentPosition);  
  
 Sequence resultSequence = \_findCycle(newSequence, Direction::***Horizontal***);  
 **if** (!resultSequence.isEmpty()) {  
 **return** resultSequence;  
 }  
 }  
  
 currentPosition.row--;  
 }  
  
 *//Проход вниз* currentPosition = sequence.getPosition(-1);  
  
 **while** (currentPosition.row <= planTable.size() - 1) {  
 **bool** isCurrentPositionInSequence = sequence.checkIfPositionInSequence(currentPosition);  
  
 **if** (getPlanTableElement(currentPosition).isBasic && !isCurrentPositionInSequence ||  
 sequence.positions.size() > 2 && currentPosition == sequence.positions[0]) {  
 Sequence newSequence = sequence;  
 newSequence.positions.push\_back(currentPosition);  
  
 Sequence resultSequence = \_findCycle(newSequence, Direction::***Horizontal***);  
 **if** (!resultSequence.isEmpty()) {  
 **return** resultSequence;  
 }  
 }  
  
 currentPosition.row++;  
 }  
 }  
  
 **if** (direction == Direction::***Horizontal*** || direction == Direction::***Any***) {  
 *//Проход влево* currentPosition = sequence.getPosition(-1);  
  
 **while** (currentPosition.column >= 0) {  
 **bool** isCurrentPositionInSequence = sequence.checkIfPositionInSequence(currentPosition);  
  
 **if** (getPlanTableElement(currentPosition).isBasic && !isCurrentPositionInSequence ||  
 sequence.positions.size() > 2 && currentPosition == sequence.positions[0]) {  
 Sequence newSequence = sequence;  
 newSequence.positions.push\_back(currentPosition);  
  
 Sequence resultSequence = \_findCycle(newSequence, Direction::***Vertical***);  
 **if** (!resultSequence.isEmpty()) {  
 **return** resultSequence;  
 }  
 }  
  
 currentPosition.column--;  
 }  
  
  
 *//Проход вправо* currentPosition = sequence.getPosition(-1);  
  
 **while** (currentPosition.column <= planTable[0].size() - 1) {  
 currentPosition.column++;  
  
 **bool** isCurrentPositionInSequence = sequence.checkIfPositionInSequence(currentPosition);  
  
 **if** (getPlanTableElement(currentPosition).isBasic && !isCurrentPositionInSequence ||  
 sequence.positions.size() > 2 && currentPosition == sequence.positions[0]) {  
 Sequence newSequence = sequence;  
 newSequence.positions.push\_back(currentPosition);  
  
 Sequence resultSequence = \_findCycle(newSequence, Direction::***Vertical***);  
 **if** (!resultSequence.isEmpty()) {  
 **return** resultSequence;  
 }  
 }  
 }  
 }  
  
 **return** {};  
 }  
}  
  
PlanTableElement TransportTable::getPlanTableElement(Position position) {  
 **return** planTable[position.row][position.column];  
}  
  
**void** TransportTable::makeShiftByCycle(Sequence cycle, **float** value) {  
 **for** (**int** i = 0; i < cycle.positions.size() - 1; i++) {  
 Position currentPosition = cycle.getPosition(i);  
 **if** (i % 2 == 0) {  
 planTable[currentPosition.row][currentPosition.column].value += value;  
 } **else** {  
 planTable[currentPosition.row][currentPosition.column].value -= value;  
 }  
 }  
}  
  
  
  
**bool** Sequence::checkIfCycle() {  
 **for** (**int** i = 0; i < positions.size() - 1; i++) {  
 **if** (positions[i].row != positions[i + 1].row &&  
 positions[i].column != positions[i + 1].column) {  
 **return false**;  
 }  
 }  
  
 **return** positions.size() > 1 && positions[0] == positions[positions.size() - 1];  
}  
  
**void** Sequence::addPosition(Position position) {  
 positions.push\_back(position);  
}  
  
Position Sequence::getPosition(**int** index) {  
 **if** (index >= 0)  
 **return** positions[index];  
 **else** {  
 **return** positions[positions.size() + index];  
 }  
}  
  
**bool** Sequence::isEmpty() {  
 **return** positions.empty();  
}  
  
**bool** Sequence::checkIfPositionInSequence(Position target) {  
 **for** (**int** i = 0; i < positions.size(); i++) {  
 **if** (positions[i] == target)  
 **return true**;  
 }  
  
 **return false**;  
}  
  
**bool** Position::**operator**==(Position other) {  
 **return this**->row == other.row && **this**->column == other.column;  
}

**Основной код распределительного метода:**

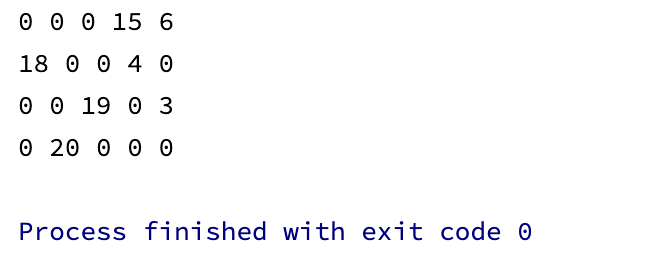
**void** TransportTable::solveByDistributiveMethod() {  
 fillTransportTableMinValue();  
  
 **bool** foundSolution = **false**;  
 **while** (!foundSolution) {  
 **float** minGamma = **INT32\_MAX**;  
 Sequence cycleWithMinValue;  
  
 **for** (**int** i = 0; i < planTable.size(); i++) {  
 **for** (**int** j = 0; j < planTable[0].size(); j++) {  
 std::cout << planTable[i][j].value << **" "**;  
 }  
  
 std::cout << **"\n"**;  
 }  
  
 std::cout << **"\n"**;  
  
 **for** (**int** row = 0; row < planTable.size(); row++) {  
 **for** (**int** column = 0; column < planTable[0].size(); column++) {  
 **if** (!planTable[row][column].isBasic) {  
 Sequence currentCycle = findCycle({row, column});  
 **float** cycleGamma = countCycleGamma(currentCycle);  
  
 **if** (cycleGamma < minGamma) {  
 minGamma = cycleGamma;  
 cycleWithMinValue = currentCycle;  
 }  
 }  
 }  
 }  
  
 **if** (minGamma < 0 && !fcmp(minGamma, 0)) {  
 **float** minAmongNegative = **INT32\_MAX**;  
 Position positionOfMinAmongNegative{-1, -1};  
   
 **for** (**int** i = 1; i < cycleWithMinValue.positions.size() - 1; i += 2) {  
 **float** currentValueWithNegativePosition = getPlanTableElement(cycleWithMinValue.getPosition(i)).value;  
 **if** (currentValueWithNegativePosition < minAmongNegative) {  
 minAmongNegative = currentValueWithNegativePosition;  
 positionOfMinAmongNegative = cycleWithMinValue.getPosition(i);  
 }  
 }  
  
 makeShiftByCycle(cycleWithMinValue, minAmongNegative);  
 planTable[positionOfMinAmongNegative.row][positionOfMinAmongNegative.column].isBasic = **false**;  
 planTable[cycleWithMinValue.getPosition(0).row][cycleWithMinValue.getPosition(0).column].isBasic = **true**;  
 } **else** {  
 foundSolution = **true**;  
 }  
 }  
}

**void** TransportTable::fillPotentialsColumn(vector<Potential> &rows, vector<Potential> &columns, **int** column) {  
 **for** (**int** row = 0; row < rows.size(); row++) {  
 **if** (planTable[row][column].isBasic && !rows[row].isSet) {  
 rows[row].value = costMatrix.getData(row, column) - columns[column].value;  
 rows[row].isSet = **true**;  
 fillPotentialsRow(rows, columns, row);  
 }  
 }  
}  
  
**void** TransportTable::fillPotentialsRow(vector<Potential> &rows, vector<Potential> &columns, **int** row) {  
 **for** (**int** column = 0; column < columns.size(); column++) {  
 **if** (planTable[row][column].isBasic && !columns[column].isSet) {  
 columns[column].value = costMatrix.getData(row, column) - rows[row].value;  
 columns[column].isSet = **true**;  
 fillPotentialsColumn(rows, columns, column);  
 }  
 }  
}

**main.cpp**

#include **<iostream>**#include **"libs/matrix/matrix.h"**#include **"libs/transport/transport.h"  
  
int** main() {  
 Matrix costMatrix;  
 costMatrix.inputMatrix(4, 5, **{** {14, 27, 6, 16, 8},  
 {2, 4, 19, 4, 27},  
 {26, 23, 1, 20, 3},  
 {24, 5, 12, 30, 5}  
 **}**);  
  
 vector<**float**> stock**{**21, 22, 22, 20**}**;  
 vector<**float**> requests**{**18, 20, 19, 19, 9**}**;  
  
 TransportTable transportTable(costMatrix, stock, requests);  
  
 transportTable.solveByDistributiveMethod();  
  
 **for** (**int** i = 0; i < transportTable.planTable.size(); i++) {  
 **for** (**int** j = 0; j < transportTable.planTable[0].size(); j++) {  
 std::cout << transportTable.planTable[i][j].value << **" "**;  
 }  
  
 std::cout << **"\n"**;  
 }  
}

**Пример вывода программы:**

****

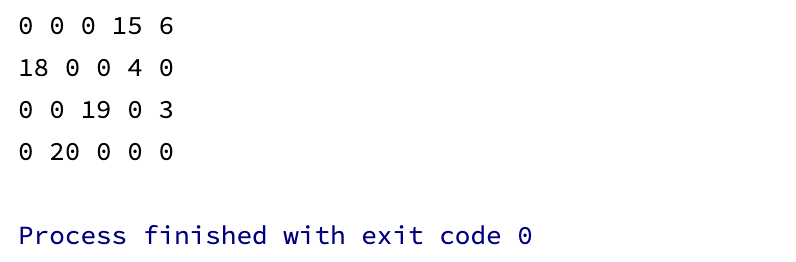
**Основной код метода потенциалов:**

**void** TransportTable::solveByPotentialMethod() {  
 fillTransportTableMinValue();  
  
 **bool** foundSolution = **false**;  
  
 **while** (!foundSolution) {  
 vector<Potential> stockPotentials(stock.size());  
 vector<Potential> requestPotentials(requests.size());  
  
 **for** (**auto** potential: stockPotentials) {  
 potential.isSet = **false**;  
 }  
  
 **for** (**auto** potential: requestPotentials) {  
 potential.isSet = **false**;  
 }  
  
 requestPotentials[0].value = 0;  
 requestPotentials[0].isSet = **true**;  
 fillPotentialsColumn(stockPotentials, requestPotentials, 0);  
  
 **float** minPotentialValue = **INT32\_MAX**;  
 Position minPotentialPosition{-1, -1};  
 **for** (**int** row = 0; row < costMatrix.getRows(); row++) {  
 **for** (**int** column = 0; column < costMatrix.getColumns(); column++) {  
 **float** currentPotential = costMatrix.getData(row, column) - (stockPotentials[row].value +  
 requestPotentials[column].value);  
 **if** (currentPotential < minPotentialValue) {  
 minPotentialValue = currentPotential;  
 minPotentialPosition = Position {row, column};  
 }  
 }  
 }  
  
 **if** (minPotentialValue < 0 && !fcmp(minPotentialValue, 0)) {  
 Sequence cycleWithMinValue = findCycle(minPotentialPosition);  
  
 **float** minAmongNegative = **INT32\_MAX**;  
 Position positionOfMinAmongNegative{-1, -1};  
  
 **for** (**int** i = 1; i < cycleWithMinValue.positions.size() - 1; i += 2) {  
 **float** currentValueWithNegativePosition = getPlanTableElement(cycleWithMinValue.getPosition(i)).value;  
 **if** (currentValueWithNegativePosition < minAmongNegative) {  
 minAmongNegative = currentValueWithNegativePosition;  
 positionOfMinAmongNegative = cycleWithMinValue.getPosition(i);  
 }  
 }  
  
 makeShiftByCycle(cycleWithMinValue, minAmongNegative);  
 planTable[positionOfMinAmongNegative.row][positionOfMinAmongNegative.column].isBasic = **false**;  
 planTable[cycleWithMinValue.getPosition(0).row][cycleWithMinValue.getPosition(0).column].isBasic = **true**;  
 } **else** {  
 foundSolution = **true**;  
 }  
 }  
}

**main.cpp**

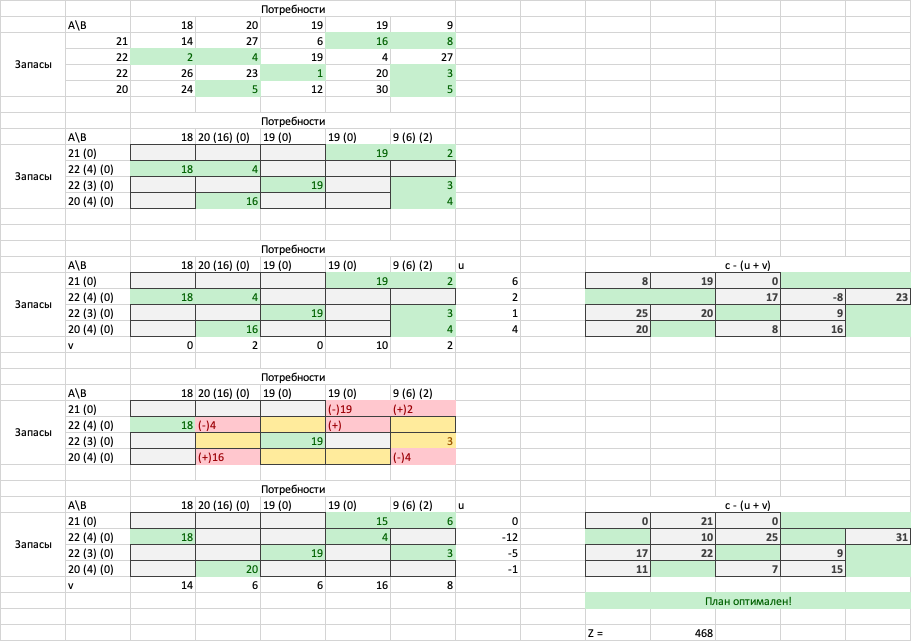
#include **<iostream>**#include **"libs/matrix/matrix.h"**#include **"libs/transport/transport.h"  
  
int** main() {  
 Matrix costMatrix;  
 costMatrix.inputMatrix(4, 5, **{** {14, 27, 6, 16, 8},  
 {2, 4, 19, 4, 27},  
 {26, 23, 1, 20, 3},  
 {24, 5, 12, 30, 5}  
 **}**);  
  
 vector<**float**> stock**{**21, 22, 22, 20**}**;  
 vector<**float**> requests**{**18, 20, 19, 19, 9**}**;  
  
 TransportTable transportTable(costMatrix, stock, requests);  
  
 transportTable.solveByPotentialMethod();  
  
 **for** (**int** i = 0; i < transportTable.planTable.size(); i++) {  
 **for** (**int** j = 0; j < transportTable.planTable[0].size(); j++) {  
 std::cout << transportTable.planTable[i][j].value << **" "**;  
 }  
  
 std::cout << **"\n"**;  
 }  
}

**Пример вывода программы:**



1. Для подготовки тестовых данных решить вручную одну из следующих ниже задач.

****



**Вывод:** в ходе лабораторной работы мы изучили методы решения закрытой транспортной задачи; реализовали заполнение исходной таблицы методом наименьшей стоимости, решение задачи методом потенциалов и распределительным методом.