НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Лабораторна робота №4

з дисципліни **«**Організація обчислювальних процесів**»**

Виконав:

студент 3 курсу

ФІОТ гр. ІО-21

Кузьменко Володимир

Перевірив:

Сімоненко В. П.

Київ – 2015 р.

**Варіант**

14 mod 13 + 1 =2

2.Скласти програму реалізації угорського методу для неоднорідної обчислювальної системи.

**Опис алгоритму**

Завдання планування зводиться до пошуку максимального паросочетання в звешенном дводольному графі, треба знайти максимальне паросочетання, щоб сума призначень була мінімальна. Венгерский алгоритм складається з наступних шагів:

1) В кожному стовпці матриці знаходимо мінімальний елемент і віднімаємо від кожного елемента цього стовпця.

2) В кожному рядку матриці знаходимо мінімальний елемент і віднімаємо від кожного елемента стовпця.

3) В кожному рядку і стовпці повинен виявитися як мінімум один «0». З цієї матриці забираємо все «0» і замінюємо на «1».

4) Знаходимо максимальнэ паросочитання. Якщо ми отримаємо досконалий варіант, то перенесем його на першу матрицю, отримаємо мінімальну суму і переходимо на пункт 16 алгоритму). Як що ні виконуємо наступний пункт.

5) На цій матриці відзначаємо нулі, які увійшли до рішення з шагу 4 і ті, які не ввійшли (закреслюємо). Відзначаємо рядки, у якіх є закреслювані «0».

6) Відмічаємо ті стовпці, в яких є закреслювані нулі, відзначених рядків.

7) Відмічаємо рядок, що містить відзначений «0», який містить рядок, відзначений в придыдущем кроці.

8) Відмічаємо стовпець, в якому є закреслюваний «0», який містить рядок, відзначений в придыдущем кроці.

9) Відмічаємо пунктиром помічені стовпці і непомічені рядки.

10) Виписуємо елементи, через які проходять пунктирні лінії і серед них шукаємо мінімальний елемент.

11) Віднімаємо цей мінімальний елемент від тих стовпців матриці через які не проходять пунктирні лінії.

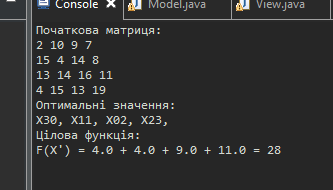
12) Додаємо цей елемент до тих рядків, через які проходять пунктирні лінії.

13) Об'еднувані одиниці переносимо на початкову матрицю.

14) Переходимо к пункту 4

15) Кінец решення.

**Результати виконання:**

****

**Лістинг програми:**

|  |
| --- |
|  |
|  | package oop.lab4; |
|  | public class Executor { |
|  |  |
|  | public static void main(String[] args) { |
|  | float [][] matr = {{2,10,9,7},{15,4,14,8},{13,14,16,11},{4,15,13,19}}; |
|  |  |
|  | float [] [] inputMatr = new float[matr.length][matr.length]; |
|  |  |
|  | for (int i = 0; i < matr.length; i++) { |
|  | for (int j = 0; j < matr.length; j++) { |
|  | inputMatr[i][j] = matr[i][j]; |
|  | } |
|  | } |
|  | int [][] res = new Hungarian(matr).execute(); |
|  |  |
|  | System.out.println("Îïòèìàëüí³ çíà÷åííÿ:"); |
|  | for (int i = 0; i < res.length; i++) { |
|  | System.out.print("X"); |
|  | for (int j = 0; j < res[i].length; j++) { |
|  | System.out.print(res[i][j]); |
|  |  |
|  | } |
|  | System.out.print(", "); |
|  | } |
|  |  |
|  |  |
|  | System.out.print("\nÖ³ëîâà ôóíêö³ÿ: \nF(X') = "); |
|  | int fx = 0; |
|  | for (int i = 0; i < res.length; i++) { |
|  | fx +=inputMatr[res[i][0]][res[i][1]]; |
|  | System.out.print(inputMatr[res[i][0]][res[i][1]]); |
|  | if (i<res.length-1){ |
|  | System.out.print(" + "); |
|  | } |
|  |  |
|  | } |
|  | System.out.print(" = "+fx); |
|  | } |
|  |  |
|  | } |

|  |
| --- |
|  |
|  | package oop.lab4; |
|  | import java.util.ArrayList; |
|  |  |
|  | public class Hungarian { |
|  |  |
|  | private int numRows; |
|  | private int numCols; |
|  |  |
|  | private boolean[][] primes; |
|  | private boolean[][] stars; |
|  | private boolean[] rowsCovered; |
|  | private boolean[] colsCovered; |
|  | private float[][] costs; |
|  |  |
|  | public Hungarian(float theCosts[][]) { |
|  | costs = theCosts; |
|  | numRows = costs.length; |
|  | numCols = costs[0].length; |
|  |  |
|  | primes = new boolean[numRows][numCols]; |
|  | stars = new boolean[numRows][numCols]; |
|  |  |
|  | // Èíèöèàëèçàöèÿ ìàññèâîâ ñ ïîêðûòèåì ñòðîê/ñòîëáöîâ |
|  | rowsCovered = new boolean[numRows]; |
|  | colsCovered = new boolean[numCols]; |
|  | for (int i = 0; i < numRows; i++) { |
|  | rowsCovered[i] = false; |
|  | } |
|  | for (int j = 0; j < numCols; j++) { |
|  | colsCovered[j] = false; |
|  | } |
|  | // Èíèöèàëèçàöèÿ ìàòðèö |
|  | for (int i = 0; i < numRows; i++) { |
|  | for (int j = 0; j < numCols; j++) { |
|  | primes[i][j] = false; |
|  | stars[i][j] = false; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | public int[][] execute() { |
|  | subtractRowColMins(); |
|  |  |
|  | this.findStars(); // O(n^2) |
|  | this.resetCovered(); // O(n); |
|  | this.coverStarredZeroCols(); // O(n^2) |
|  |  |
|  | while (!allColsCovered()) { |
|  | int[] primedLocation = this.primeUncoveredZero(); // O(n^2) |
|  | if (primedLocation[0] == -1) { |
|  | this.minUncoveredRowsCols(); // O(n^2) |
|  | primedLocation = this.primeUncoveredZero(); // O(n^2) |
|  | } |
|  |  |
|  | // is there a starred 0 in the primed zeros row? |
|  | int primedRow = primedLocation[0]; |
|  | int starCol = this.findStarColInRow(primedRow); |
|  | if (starCol != -1) { |
|  | // cover ther row of the primedLocation and uncover the star column |
|  | rowsCovered[primedRow] = true; |
|  | colsCovered[starCol] = false; |
|  | } else { // otherwise we need to find an augmenting path and start over. |
|  | this.augmentPathStartingAtPrime(primedLocation); |
|  | this.resetCovered(); |
|  | this.resetPrimes(); |
|  | this.coverStarredZeroCols(); |
|  | } |
|  | } |
|  |  |
|  | return this.starsToAssignments(); // O(n^2) |
|  |  |
|  | } |
|  |  |
|  | public int[][] starsToAssignments() { |
|  | int[][] toRet = new int[numCols][]; |
|  | for (int j = 0; j < numCols; j++) { |
|  | toRet[j] = new int[] { |
|  | this.findStarRowInCol(j), j |
|  | }; // O(n) |
|  | } |
|  | return toRet; |
|  | } |
|  | public void resetPrimes() { |
|  | for (int i = 0; i < numRows; i++) { |
|  | for (int j = 0; j < numCols; j++) { |
|  | primes[i][j] = false; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | public void resetCovered() { |
|  | for (int i = 0; i < numRows; i++) { |
|  | rowsCovered[i] = false; |
|  | } |
|  | for (int j = 0; j < numCols; j++) { |
|  | colsCovered[j] = false; |
|  | } |
|  | } |
|  | public void findStars() { |
|  | boolean[] rowStars = new boolean[numRows]; |
|  | boolean[] colStars = new boolean[numCols]; |
|  |  |
|  | for (int i = 0; i < numRows; i++) { |
|  | rowStars[i] = false; |
|  | } |
|  | for (int j = 0; j < numCols; j++) { |
|  | colStars[j] = false; |
|  | } |
|  |  |
|  | for (int j = 0; j < numCols; j++) { |
|  | for (int i = 0; i < numRows; i++) { |
|  | if (costs[i][j] == 0 && !rowStars[i] && !colStars[j]) { |
|  | stars[i][j] = true; |
|  | rowStars[i] = true; |
|  | colStars[j] = true; |
|  | break; |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | private void minUncoveredRowsCols() { |
|  | // find min uncovered value |
|  | float minUncovered = Float.MAX\_VALUE; |
|  | for (int i = 0; i < numRows; i++) { |
|  | if (!rowsCovered[i]) { |
|  | for (int j = 0; j < numCols; j++) { |
|  | if (!colsCovered[j]) { |
|  | if (costs[i][j] < minUncovered) { |
|  | minUncovered = costs[i][j]; |
|  | } |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | // add that value to all the COVERED rows. |
|  | for (int i = 0; i < numRows; i++) { |
|  | if (rowsCovered[i]) { |
|  | for (int j = 0; j < numCols; j++) { |
|  | costs[i][j] = costs[i][j] + minUncovered; |
|  |  |
|  | } |
|  | } |
|  | } |
|  |  |
|  | // subtract that value from all the UNcovered columns |
|  | for (int j = 0; j < numCols; j++) { |
|  | if (!colsCovered[j]) { |
|  | for (int i = 0; i < numRows; i++) { |
|  | costs[i][j] = costs[i][j] - minUncovered; |
|  | } |
|  | } |
|  | } |
|  | } |
|  | private int[] primeUncoveredZero() { |
|  | int[] location = new int[2]; |
|  |  |
|  | for (int i = 0; i < numRows; i++) { |
|  | if (!rowsCovered[i]) { |
|  | for (int j = 0; j < numCols; j++) { |
|  | if (!colsCovered[j]) { |
|  | if (costs[i][j] == 0) { |
|  | primes[i][j] = true; |
|  | location[0] = i; |
|  | location[1] = j; |
|  | return location; |
|  | } |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | location[0] = -1; |
|  | location[1] = -1; |
|  | return location; |
|  | } |
|  |  |
|  | private void augmentPathStartingAtPrime(int[] location) { |
|  | // Make the arraylists sufficiently large to begin with |
|  | ArrayList < int[] > primeLocations = new ArrayList < int[] > (numRows + numCols); |
|  | ArrayList < int[] > starLocations = new ArrayList < int[] > (numRows + numCols); |
|  | primeLocations.add(location); |
|  |  |
|  | int currentRow = location[0]; |
|  | int currentCol = location[1]; |
|  | while (true) { // add stars and primes in pairs |
|  | int starRow = findStarRowInCol(currentCol); |
|  | if (starRow == -1) { |
|  | break; |
|  | } |
|  | int[] starLocation = new int[] { |
|  | starRow, currentCol |
|  | }; |
|  | starLocations.add(starLocation); |
|  | currentRow = starRow; |
|  |  |
|  | int primeCol = findPrimeColInRow(currentRow); |
|  | int[] primeLocation = new int[] { |
|  | currentRow, primeCol |
|  | }; |
|  | primeLocations.add(primeLocation); |
|  | currentCol = primeCol; |
|  | } |
|  |  |
|  | unStarLocations(starLocations); |
|  | starLocations(primeLocations); |
|  | } |
|  |  |
|  |  |
|  | private void starLocations(ArrayList < int[] > locations) { |
|  | for (int k = 0; k < locations.size(); k++) { |
|  | int[] location = locations.get(k); |
|  | int row = location[0]; |
|  | int col = location[1]; |
|  | stars[row][col] = true; |
|  | } |
|  | } |
|  |  |
|  | private void unStarLocations(ArrayList < int[] > starLocations) { |
|  | for (int k = 0; k < starLocations.size(); k++) { |
|  | int[] starLocation = starLocations.get(k); |
|  | int row = starLocation[0]; |
|  | int col = starLocation[1]; |
|  | stars[row][col] = false; |
|  | } |
|  | } |
|  |  |
|  | private int findPrimeColInRow(int theRow) { |
|  | for (int j = 0; j < numCols; j++) { |
|  | if (primes[theRow][j]) { |
|  | return j; |
|  | } |
|  | } |
|  | return -1; |
|  | } |
|  |  |
|  | public int findStarRowInCol(int theCol) { |
|  | for (int i = 0; i < numRows; i++) { |
|  | if (stars[i][theCol]) { |
|  | return i; |
|  | } |
|  | } |
|  | return -1; |
|  | } |
|  |  |
|  |  |
|  | public int findStarColInRow(int theRow) { |
|  | for (int j = 0; j < numCols; j++) { |
|  | if (stars[theRow][j]) { |
|  | return j; |
|  | } |
|  | } |
|  | return -1; |
|  | } |
|  | private boolean allColsCovered() { |
|  | for (int j = 0; j < numCols; j++) { |
|  | if (!colsCovered[j]) { |
|  | return false; |
|  | } |
|  | } |
|  | return true; |
|  | } |
|  | private void coverStarredZeroCols() { |
|  | for (int j = 0; j < numCols; j++) { |
|  | colsCovered[j] = false; |
|  | for (int i = 0; i < numRows; i++) { |
|  | if (stars[i][j]) { |
|  | colsCovered[j] = true; |
|  | break; // break inner loop to save a bit of time |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | private void subtractRowColMins() { |
|  | for (int i = 0; i < numRows; i++) { //for each row |
|  | float rowMin = Float.MAX\_VALUE; |
|  | for (int j = 0; j < numCols; j++) { // grab the smallest element in that row |
|  | if (costs[i][j] < rowMin) { |
|  | rowMin = costs[i][j]; |
|  | } |
|  | } |
|  | for (int j = 0; j < numCols; j++) { // subtract that from each element |
|  | costs[i][j] = costs[i][j] - rowMin; |
|  | } |
|  | } |
|  |  |
|  | for (int j = 0; j < numCols; j++) { // for each col |
|  | float colMin = Float.MAX\_VALUE; |
|  | for (int i = 0; i < numRows; i++) { // grab the smallest element in that column |
|  | if (costs[i][j] < colMin) { |
|  | colMin = costs[i][j]; |
|  | } |
|  | } |
|  | for (int i = 0; i < numRows; i++) { // subtract that from each element |
|  | costs[i][j] = costs[i][j] - colMin; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | } |