Kpi-best

Міністерство освіти та науки, молоді та спорту України

Національний технічний університет України «Київський політехнічний інститут»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Лабораторна робота № 4

з курсу «Системне програмне забезпечення - 1» на тему:

«**Дослідження принципів проектування та роботи динамічного та статичного планування**»

Виконала: студентка 3 курсу

ФІОТ, гр. ІО - 02

Агеєнко Юлія

Київ

2013 рік

## Завдання на роботу

Написати програму, яка реалізує один з алгоритмів динамічного або статичного планування згідно варіанту.

## Варіанти

Варіант вибирається по номеру залікової книжки: mod 13 + 1

№ залікової книжки: 222

1. d 13) + 1 = 1 + 1 = 2

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

Для вирішення завдання розподілу завдань по ресурсах в неоднорідній обчислювальній системі можна використовувати угорський алгоритм пошуку максимального паросполучення з використанням теорем 1-5 .

## Угорський метод.

Завдання планування зводиться до пошуку максимального паросполучення в зваженому дводольному графі, треба знайти максимальне паросполучення, щоб сума призначень була мінімальна. Угорський алгоритм складається з наступних кроків:

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

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

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

4) Знаходимо максимальне паросполучення. Якщо ми отримаємо досконалий варіант, то перенесемо його на першу матрицю, отримаємо мінімальну суму і перейдемо на пункт 16 алгоритму). Якщо ні, виконуємо наступний пункт.

5) На цій матриці відзначаємо нулі, які увійшли до розв’язку з шагу 4 і ті, які не ввійшли (закреслюємо). Відзначаємо рядки, у яких 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 попередньому кроці.

8) Відмічаємо стовпець, в якому є закреслюваний «0», який містить рядок, відзначений в попередньому 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’єднувані одиниці переносимо на початкову матрицю.

14) Переходимо к пункту 4

15) Кінець розв’язку.

**Лістинг основної частини програми.**

**public** **class** HungarianAlgorithm {

**public** **static** **double**[][] copyOf //Copies all elements of an array to a new array.

(**double**[][] original){

**int** rows = original.length;

**int** cols = original[0].length;

**double**[][] copy = **new** **double**[original.length][original[0].length];

**if** (rows == cols){ //The matrix is already square.

**for** (**int** i=0; i<original.length; i++){

//Need to do it this way, otherwise it copies only memory location

System.*arraycopy*(original[i], 0, copy[i], 0, original[i].length);

}

}

**else** {

System.*out*.println("The matrix is not square!");

}

**return** copy;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//METHODS OF THE HUNGARIAN ALGORITHM//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//Core of the algorithm; takes required inputs and returns the assignments

**public** **static** **int**[][] hgAlgorithmAssignments(**double**[][] array){

//This variable is used to pad a rectangular array (so it will be picked all last [cost] or first [profit])

//and will not interfere with final assignments. Also, it is used to flip the relationship between weights

//when "max" defines it as a profit matrix instead of a cost matrix. Double.MAX\_VALUE is not ideal, since arithmetic

//needs to be performed and overflow may occur.

**double**[][] cost = *copyOf*(array); //Create the cost matrix

**int**[][] mask = **new** **int**[cost.length][cost[0].length]; //The mask array.

**int**[] rowCover = **new** **int**[cost.length]; //The row covering vector.

**int**[] colCover = **new** **int**[cost[0].length]; //The column covering vector.

**int**[] zero\_RC = **new** **int**[2]; //Position of last zero from Step 4.

**int**[][] path = **new** **int**[cost.length \* cost[0].length + 2][2];

**int** step = 1;

**boolean** done = **false**;

**while** (done == **false**){ //main execution loop

**switch** (step){

**case** 1:

step = *hg\_step1*(step, cost);

**break**;

**case** 2:

step = *hg\_step2*(step, cost, mask, rowCover, colCover);

**break**;

**case** 3:

step = *hg\_step3*(step, mask, colCover);

**break**;

**case** 4:

step = *hg\_step4*(step, cost, mask, rowCover, colCover, zero\_RC);

**break**;

**case** 5:

step = *hg\_step5*(step, mask, rowCover, colCover, zero\_RC, path);

**break**;

**case** 6:

step = *hg\_step6*(step, cost, rowCover, colCover);

**break**;

**case** 7:

done=**true**;

**break**;

}

}//end while

**int**[][] assignments = **new** **int**[array.length][2]; //Create the returned array.

**int** assignmentCount = 0; //In a input matrix taller than it is wide, the first

//assignments column will have to skip some numbers, so

//the index will not always match the first column ([0])

**for** (**int** i=0; i<mask.length; i++){

**for** (**int** j=0; j<mask[i].length; j++){

**if** (i < array.length && j < array[0].length && mask[i][j] == 1){

assignments[assignmentCount][0] = i;

assignments[assignmentCount][1] = j;

assignmentCount++;

}

}

}

**return** assignments;

}

//Calls hgAlgorithmAssignments and getAssignmentSum to compute the

//minimum cost possible.

**public** **static** **double** hgAlgorithm(**double**[][] array){

**return** *getAssignmentSum*(array, *hgAlgorithmAssignments*(array));

}

**public** **static** **double** getAssignmentSum(**double**[][] array, **int**[][] assignments) {

//Returns the min/max sum (cost/profit of the assignment) given the

//original input matrix and an assignment array (from hgAlgorithmAssignments)

**double** sum = 0;

**for** (**int** i=0; i<assignments.length; i++){

**if** (i==assignments.length-1) {

System.*out*.print(array[assignments[i][0]][assignments[i][1]]);

}**else** {

System.*out*.print(array[assignments[i][0]][assignments[i][1]]+" + ");

}

sum = sum + array[assignments[i][0]][assignments[i][1]];

}

System.*out*.print(" = ");

**return** sum;

}

**public** **static** **int** hg\_step1(**int** step, **double**[][] cost){

//What STEP 1 does:

//For each row of the cost matrix, find the smallest element

//and subtract it from from every other element in its row.

**double** minval;

**for** (**int** i=0; i<cost.length; i++){

minval=cost[i][0];

**for** (**int** j=0; j<cost[i].length; j++){ //1st inner loop finds min val in row.

**if** (minval>cost[i][j]){

minval=cost[i][j];

}

}

**for** (**int** j=0; j<cost[i].length; j++){ //2nd inner loop subtracts it.

cost[i][j]=cost[i][j]-minval;

// System.out.print(cost[i][j] + " ");

}

// System.out.println(" ");

}

step = 2;

**return** step;

}

**public** **static** **int** hg\_step2(**int** step, **double**[][] cost, **int**[][] mask, **int**[] rowCover, **int**[] colCover){

//What STEP 2 does:

//Marks uncovered zeros as starred and covers their row and column.

**for** (**int** i=0; i<cost.length; i++){

**for** (**int** j=0; j<cost[i].length; j++){

**if** ((cost[i][j]==0) && (colCover[j]==0) && (rowCover[i]==0)){

mask[i][j]=1;

colCover[j]=1;

rowCover[i]=1;

}

}

}

*clearCovers*(rowCover, colCover); //Reset cover vectors.

step = 3;

**return** step;

}

**public** **static** **int** hg\_step3(**int** step, **int**[][] mask, **int**[] colCover){

//What STEP 3 does:

//Cover columns of starred zeros. Check if all columns are covered.

**for** (**int** i=0; i<mask.length; i++){ //Cover columns of starred zeros.

**for** (**int** j=0; j<mask[i].length; j++){

**if** (mask[i][j] == 1){

colCover[j]=1;

}

}

}

**int** count=0;

**for** (**int** j=0; j<colCover.length; j++){ //Check if all columns are covered.

count=count+colCover[j];

}

**if** (count>=mask.length){ //Should be cost.length but ok, because mask has same dimensions.

step = 7;

}

**else** {

step = 4;

}

**return** step;

}

**public** **static** **int** hg\_step4(**int** step, **double**[][] cost, **int**[][] mask, **int**[] rowCover, **int**[] colCover, **int**[] zero\_RC){

//What STEP 4 does:

//Find an uncovered zero in cost and prime it (if none go to step 6). Check for star in same row:

//if yes, cover the row and uncover the star's column. Repeat until no uncovered zeros are left

//and go to step 6. If not, save location of primed zero and go to step 5.

**int**[] row\_col = **new** **int**[2]; //Holds row and col of uncovered zero.

**boolean** done = **false**;

**while** (done == **false**){

row\_col = *findUncoveredZero*(row\_col, cost, rowCover, colCover);

**if** (row\_col[0] == -1){

done = **true**;

step = 6;

}

**else** {

mask[row\_col[0]][row\_col[1]] = 2; //Prime the found uncovered zero.

**boolean** starInRow = **false**;

**for** (**int** j=0; j<mask[row\_col[0]].length; j++){

**if** (mask[row\_col[0]][j]==1){ //If there is a star in the same row...

starInRow = **true**;

row\_col[1] = j; //remember its column.

}

}

**if** (starInRow==**true**){

rowCover[row\_col[0]] = 1; //Cover the star's row.

colCover[row\_col[1]] = 0; //Uncover its column.

}

**else** {

zero\_RC[0] = row\_col[0]; //Save row of primed zero.

zero\_RC[1] = row\_col[1]; //Save column of primed zero.

done = **true**;

step = 5;

}

}

}

**return** step;

}

**public** **static** **int**[] findUncoveredZero //Aux 1 for hg\_step4.

(**int**[] row\_col, **double**[][] cost, **int**[] rowCover, **int**[] colCover){

row\_col[0] = -1; //Just a check value. Not a real index.

row\_col[1] = 0;

**int** i = 0; **boolean** done = **false**;

**while** (done == **false**){

**int** j = 0;

**while** (j < cost[i].length){

**if** (cost[i][j]==0 && rowCover[i]==0 && colCover[j]==0){

row\_col[0] = i;

row\_col[1] = j;

done = **true**;

}

j = j+1;

}//end inner while

i=i+1;

**if** (i >= cost.length){

done = **true**;

}

}//end outer while

**return** row\_col;

}

**public** **static** **int** hg\_step5(**int** step, **int**[][] mask, **int**[] rowCover, **int**[] colCover, **int**[] zero\_RC, **int** [][] path){

//What STEP 5 does:

//Construct series of alternating primes and stars. Start with prime from step 4.

//Take star in the same column. Next take prime in the same row as the star. Finish

//at a prime with no star in its column. Unstar all stars and star the primes of the

//series. Erasy any other primes. Reset covers. Go to step 3.

**int** count = 0; //Counts rows of the path matrix.

//int[][] path = new int[(mask[0].length + 2)][2]; //Path matrix (stores row and col).

path[count][0] = zero\_RC[0]; //Row of last prime.

path[count][1] = zero\_RC[1]; //Column of last prime.

**boolean** done = **false**;

**while** (done == **false**){

**int** r = *findStarInCol*(mask, path[count][1]);

**if** (r>=0){

count = count+1;

path[count][0] = r; //Row of starred zero.

path[count][1] = path[count-1][1]; //Column of starred zero.

}

**else** {

done = **true**;

}

**if** (done == **false**){

**int** c = *findPrimeInRow*(mask, path[count][0]);

count = count+1;

path[count][0] = path[count-1][0]; //Row of primed zero.

path[count][1] = c; //Col of primed zero.

}

}//end while

*convertPath*(mask, path, count);

*clearCovers*(rowCover, colCover);

*erasePrimes*(mask);

step = 3;

**return** step;

}

**public** **static** **int** findStarInCol //Aux 1 for hg\_step5.

(**int**[][] mask, **int** col){

**int** r = -1; //Again this is a check value.

**for** (**int** i=0; i<mask.length; i++){

**if** (mask[i][col]==1){

r = i;

}

}

**return** r;

}

**public** **static** **int** findPrimeInRow //Aux 2 for hg\_step5.

(**int**[][] mask, **int** row){

**int** c = -1;

**for** (**int** j=0; j<mask[row].length; j++){

**if** (mask[row][j]==2){

c = j;

}

}

**return** c;

}

**public** **static** **void** convertPath //Aux 3 for hg\_step5.

(**int**[][] mask, **int**[][] path, **int** count){

**for** (**int** i=0; i<=count; i++){

**if** (mask[path[i][0]][path[i][1]]==1){

mask[path[i][0]][path[i][1]] = 0;

}

**else** {

mask[path[i][0]][path[i][1]] = 1;

}

}

}

**public** **static** **void** erasePrimes //Aux 4 for hg\_step5.

(**int**[][] mask){

**for** (**int** i=0; i<mask.length; i++){

**for** (**int** j=0; j<mask[i].length; j++){

**if** (mask[i][j]==2){

mask[i][j] = 0;

}

}

}

}

**public** **static** **void** clearCovers //Aux 5 for hg\_step5 (and not only).

(**int**[] rowCover, **int**[] colCover){

**for** (**int** i=0; i<rowCover.length; i++){

rowCover[i] = 0;

}

**for** (**int** j=0; j<colCover.length; j++){

colCover[j] = 0;

}

}

**public** **static** **int** hg\_step6(**int** step, **double**[][] cost, **int**[] rowCover, **int**[] colCover){

//What STEP 6 does:

//Find smallest uncovered value in cost: a. Add it to every element of covered rows

//b. Subtract it from every element of uncovered columns. Go to step 4.

**double** minval = *findSmallest*(cost, rowCover, colCover);

**for** (**int** i=0; i<rowCover.length; i++){

**for** (**int** j=0; j<colCover.length; j++){

**if** (rowCover[i]==1){

cost[i][j] = cost[i][j] + minval;

}

**if** (colCover[j]==0){

cost[i][j] = cost[i][j] - minval;

}

}

}

step = 4;

**return** step;

}

**public** **static** **double** findSmallest //Aux 1 for hg\_step6.

(**double**[][] cost, **int**[] rowCover, **int**[] colCover){

**double** minval = Double.*POSITIVE\_INFINITY*; //There cannot be a larger cost than this.

**for** (**int** i=0; i<cost.length; i++){ //Now find the smallest uncovered value.

**for** (**int** j=0; j<cost[i].length; j++){

**if** (rowCover[i]==0 && colCover[j]==0 && (minval > cost[i][j])){

minval = cost[i][j];

}

}

}

**return** minval;

}

**public** **static** **void** set(**double** [][] arr, **int** i, **int** j, **double** v) {arr[i][j] = v;}

//\*\*\*\*\*\*\*\*\*\*\*//

//MAIN METHOD//

//\*\*\*\*\*\*\*\*\*\*\*//

**public** **static** **void** main(String[] args) {

**int** M = Integer.*MAX\_VALUE*;

// 4x4

**double**[][] test1 = {{1,7,1,3},

{1,6,4,6},

{17,1,5,1},

{1,6,10,4}};

// 5x5

**double**[][] test2 = {{ 3,14,5, 6, 7},

{ 1, 9,7, 4, 3},

{ 8, 7,4, 6, 3},

{ 5, 6,9,13,15},

{19,14,16,8,12}};

// 6x6

**double**[][] test3 = {{ 8, 5, M, 7, 6, 9},

{ 6, 3, 9, 1, M, 7},

{ 5, M, 3, 6, 8, 8},

{ 9, 1, 8, M, 5, 4},

{ 1, 2, 1, M, 2, 3},

{ M, M, 3, 5, 3, 10}};

System.*out*.print("Calculation for 4x4 matrix: ");

System.*out*.println(*hgAlgorithm*(test1));

System.*out*.print("Calculation for 5x5 matrix: ");

System.*out*.println(*hgAlgorithm*(test2));

System.*out*.print("Calculation for 6x6 matrix: ");

System.*out*.println(*hgAlgorithm*(test3));

}

}

**Вивод програми:**

Calculation for 4x4 matrix: 1.0 + 1.0 + 1.0 + 4.0 = 7.0

Calculation for 5x5 matrix: 5.0 + 1.0 + 3.0 + 6.0 + 8.0 = 23.0

Calculation for 6x6 matrix: 5.0 + 1.0 + 3.0 + 4.0 + 1.0 + 3.0 = 17.0