Misr University for Science & Technology

Information Technology (CS)

Algorithm Project

CS-331

The Assignment Problem

(Using Java)

Name: Mina Romany Awad

ID: 89413

Introduction

The assignment problem, one of the fundamental optimization problems, is a linear programming model which is arranged to match the resources (employee, machine etc.) with varying tasks. The key idea is to get maximum profit or sale and to get minimum cost.

Suppose we have m workers and n machines. We know the cost of assigning machines to workers. We aimed to make an assignment which minimizes the cost. The optimal assignment is the assignment that minimizes the cost. It is accepted that the number of workers is equal to the number of machines in the assignment problem (m=n) (Balanced Matrix).

Solving the assignment problem as a transportation problem is complicated and time consuming. Thus, new methods have been developed for the assignment problem, some of the classical algorithms developed for the assignment problem are Branch Boundary Algorithm, Brute Force Algorithm, and Hungarian Algorithm. But now I will focus only on two algorithms to solve this problem (Hungarian Algorithm and Brute Force Algorithm).

Using => Java (NetBeans-IDE)

Solve Assignment Problem using :

1. Hungarian Algorithm
2. Exhaustive Search (brute force)

Project on GitHub

https://github.com/MinaRomany53/Assignment

---

1. Hungarian Algorithm

1.1. Overview

The Hungarian algorithm consists of the four steps below. The first two steps are executed once, while Steps 3 and 4 are repeated until an optimal assignment is found. The input of the algorithm is an n x n square matrix with only nonnegative elements.

The given steps are applied to the nxn cost matrix to find the optimal solution:

Step 1: The smallest element in each row is subtracted from all elements in the row.
Step 2: The smallest element in each column is subtracted from all elements in the column.
Step 3: A minimum number of lines vertical and horizontal are drawn to cover all Zeros in the cost matrix, then If (number of lines = size of matrix n) algorithm stops. Else, go to step-4.
Step 4: Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Return to Step-3.

1.2 Pseudo code

1. Helpful Methods

```
displayMatrix(M[n][n])
// Display all elements in a Matrix nxn
// Input: Balanced matrix M of size nxn
// Output: Print all elements in the matrix
        for i <--- 0 to n-1 do
            for j <--- 0 to n-1 do
                print M[i][j]
        print " " // make space after every Row
```

```
saveFirstmatrix(M[][] , M_copied[n][n])
// Copy all Original matrix M to another matrix
// Input: two matrices of size nxn
// Output: return the copy matrix
        for i <--- 0 to n-1 do
            for j <--- 0 to n-1 do
                M_copied[i][j] <--- M[i][j];

        return M_copied;
```

```
calcLines (rowLines[] , colLines[])
// count all elements are "true" in both boolean arrays
// Input: Two boolean arrays of size n
// Output: counter for true elements
        linesCounter <--- 0
        for i <--- 0 to n-1 do
            if rowLines[i] = true do
                linesCounter <--- linesCounter +1
            if colLines[i] = true
                linesCounter <--- linesCounter +1

        return linesCounter;
```

2. Hungarian Algorithm Methods

```
subRowmin(M[n][n])
// finds the lowest element in each row and subtract it from all elements in that row
// Input: Balanced Matrix (M) size nxn
// Output: new Balanced Matrix (M) size nxn
        for i <--- 0 to n-1 do
            minNumber <--- M[i][0]
            for j <--- 0 to n-1 do
                if (M[i][j] < minNumber do
                    minNumber <--- M[i][j]
            for j <--- 0 to n-1 do
                M[i][j] <--- M[i][j] - minNumber;

        return M;
```

```
subColmin(M[n][n])
// finds the lowest element in each column and subtract it from all elements in that column
// Input: Balanced Matrix (M) size nxn
// Output: new Balanced Matrix (M) size nxn
        for j <--- 0 to n-1 do
            minNumber <--- M[0][j]
            for i <--- 0 to n-1 do
                if (M[i][j] < minNumber do
                    minNumber <--- M[i][j]
            for i <--- 0 to n-1 do
                M[i][j] <--- M[i][j] - minNumber;

        return M;
```

```
drawZeroslines(M[n][n])
//Cover all zeros in matrix (M) with minimum number of lines
//Store column index of zero in solution array
//Input: Balanced Matrix of size nxn
//Output: Solution Array of size n
        solution[n] , bool rowLines[n] , bool colLines[n]
        //Check Row Zeros (one zero in a row = vertical line)
        for i <--- 0 to n-1 do
            countZeros <--- 0  , colNo <--- 0
            for j <--- 0 to n-1 do
                if M[i][j] = 0 and colLines[j] = false do
                    countZeros <--- countZeros + 1
                    colNo <--- j
            if countZeros = 1 and colLines[colNo] = false do
                colLines[colNo] = true
                solution[i] = colNo

        //check col zeros (one zero in a col = horizontal line)
        for j <--- 0 to n-1 do
            countZeros <--- 0  , rowNo <--- 0
            for i <--- 0 to n-1 do
                if M[i][j] = 0 and colLines[j] = false do
                    countZeros <--- countZeros + 1
                    rowNo <--- i
            if countZeros = 1 and rowLines[rowNo] = false do
                rowLines[rowNo] = true
                solution[rowNo] = j

        // Check last time that no zeros left uncovered
        for i <--- 0 to n-1 do
            for j <--- 0 to n-1 do
                if M[i][j] = 0 and colLines[j] = false and rowLines[j] = false do
                    rowLines[i] = true
                    solution[i] = j
```

```
// Check last time that no zeros left uncovered
for i <--- 0 to n-1 do
    for j <--- 0 to n-1 do
        if M[i][j] = 0 and colLines[j] = false and rowLines[j] = false do
            rowLines[i] = true
            solution[i] = j

// get total number of lines
noLines <--- calcLines(rowLines,colLines)

if noLines  = n do
    return solution
else
    //Create additional zeros
    //Find the smallest uncovered number
    minNumber <--- Integer.max_Value
    for i <--- 0 to n-1 do
        for j <--- 0 to n-1 do
            if colLines = false and rowLines = false and M[i][j] < minNumber
                minNumber <--- M[i][j]

    // Subtract this number from all uncovered elements
    for i <--- 0 to n-1 do
        for j <--- 0 to n-1 do
            if colLines[j] = false and rowLines[i] = false do
                M[i][j] <--- M[i][j] - minNumber

    // Add it to all elements that are covered twice
    for i <--- 0 to n-1 do
        for j <--- 0 to n-1 do
            if colLines[j] = true and rowLines[i] = true do
                M[i][j] <--- M[i][j] + minNumber


    return drawZeroslines(M)
```

3. Main Method

```
main()
// solve Assignment Problem using Hungarian Algorithm
// calling all hungarian methods here to execute it and display the solution
// Input: size of the balanced matrix nxn and its elements
// Output: display solution and the minimum optimal solutioln

        Matrix <--- input from user

        // Save a copy of the Original Matrix
        firstMatrix <--- savirseFtmatrix(matrix,firstMatrix)

        // Step-1: Subtract Row minimum
        subRowmin(matrix);

        // Step-2: Subtract Col minimum
        subColmin(matrix);

        // Step-3: Cover all zeros with a minimum number of lines
        solution <--- drawZeroslines(matrix)

        //Display Solution
        sumJobsvalues <--- 0;
        for i <--- 0 to n-1
            sumJobsvalues <---- sumJobsvalues + firstMatrix[i][solution[i]]
        print sumJobsvalues
```

1.3 Code description

1. Helpful Methods

displayMatrix() - Display the matrix update after every step in the program.

saveFirstmatrix() - Copy original matrix values to another matrix and save it because I will need the original matrix at the end to display minimum optimal solution.

calcLines() - Calculate number of lines drawing on the matrix then the return value from this method determines if we need (step 4) or not.

```java
package assignment.problem;
 // @author Mina Romany
import java.util.Scanner;
public class AssignmentProblem {
    static Scanner input = new Scanner(System.in);
    /*--------------------------------------- Helpful Methods -----------------------------------------*/
    // Display Matrix  O(n2)
    static void displayMatrix(int M[][] ) {
        for (int i = 0; i < M.length; i++) {
            for (int j = 0; j < M[i].length; j++) {
                System.out.print(M[i][j]+ " ");
            }
            System.out.println(" ");
        }
        System.out.println(" ");
    }

    // Copy Matrix O(n2)
    static int[][] saveFirstmatrix(int M[][] , int M_copied[][] ) {
        for (int i = 0; i < M.length; i++) {
            for (int j = 0; j < M[i].length; j++) {
                M_copied[i][j] = M[i][j];
            }
        }
        return M_copied;
    }

    // Calculate number of Lines drawing (horizontal or vertical) O(n)
    static int calcLines (boolean rowLines[],boolean colLines[]){
        int linesCounter = 0;
        for (int i = 0; i <rowLines.length ; i++) {
            if (rowLines[i] == true ) {linesCounter++;}
            if (colLines[i] == true ) {linesCounter++;}
        }
        return linesCounter;
    }
    /*---------------------------------------------------------------------------------------------*/
```

2. Hungarian Algorithm Methods

subRowmin() - implement **(step 1)** in the algorithm, this method finds the lowest element in each row and subtract it from all elements in that row then return new matrix.

subColmin() - implement **(step 2)** in the algorithm, this method finds the lowest element in each column and subtract it from all elements in that column then return new matrix.

```java

    /*--------------------------------------- Hungarian Algorithm Methods --------------------------------------*/
    // Subtract row minimum O(2n**2)
    static int[][] subRowmin (int M[][]){
        int n = M.length;
        for (int i = 0; i < n; i++) {
                    int minNumber = M[i][0]; // minNumber = first num in each row
            // get minimum number in the Row
            for (int j = 0; j < n; j++) {
                if (M[i][j]<minNumber ){minNumber = M[i][j];}
            }
            // Subtract it from all elements in the Row
            for (int j = 0; j < n; j++) {
                M[i][j] = M[i][j] - minNumber;
            }
        }
        return M;
    }

    // Subtract col minimum O(2n**2)
    static int[][] subColmin (int M[][]){
        int n = M.length;
        for (int j = 0; j< n; j++) {
                    int minNumber = M[0][j]; // minNumber = first num in each col
            // get minimum number in the col
            for (int i = 0; i < n; i++) {
                if (M[i][j]<minNumber ){minNumber = M[i][j];}
            }
            // Subtract it from all elements in the col
            for (int i = 0; i < n; i++) {
               M[i][j] = M[i][j] - minNumber;
            }
        }
        return M;
    }

```

drawZeroslines() -implement **(step 3)** in the algorithm, at the first part in this method it Covers all zeros in the matrix using a minimum number of horizontal and vertical lines.

1) loop over all rows if the row contains only one zero then draw a vertical line on the column (colLines[col] = true) (one zero in a row = vertical line) then store column index in solution array.

2) loop over all columns if the col contains only one zero then draw a horizontal line on the row (rowLines[row] = true) (one zero in a columns = horizontal line) then store column index in solution array.

3) check last time that no zeros left uncovered

the second part in this method, check if the number of lines (horizontal and vertical) equal to size of the matrix(n) then the method stops and return solution array that contains indexes for the optimal solution and algorithm steps are finished.

* Compute number of lines by calling calcLines() method from Helpful methods

Else continue with **(Step 4)** in this algorithm, that try to create additional zeros to make number of lines equal to size of the matrix, at first Find the smallest uncovered number then Subtract this number from all uncovered elements and add it to all elements that are covered twice.

At the last after creating additional zero recall this function recursively to Covers all zeros in the matrix using a minimum number of horizontal and vertical lines.

If (noLines == n) Initial condition to stop calling the function recursively

```java
// Cover all zeros in a Matrix with a minimum number of lines (horizontal or vertical) + assign one job to one person  O(n**2)
static int[] drawZeroslines (int M[][]) {
    int n = M.length;
    int solution[] = new int [n]; // store person(index) Assign to Job(element)
    boolean rowLines [] = new boolean [n]; // RowLines all False       ex. if n= 4 [r1,r2,r3,r4] 4
    boolean colLines [] = new boolean [n]; // ColLines all False       ex. if n= 4 [c1,c2,c3,c4] 4
    // check row zeros (Vertical lines) (one zero in a row = vertical line)
    for (int i = 0; i < n; i++) {
        int countZeros = 0;
        int colNo = 0; // store col number to make vertical line
        for (int j = 0; j < n; j++) {
            if(M[i][j] == 0 && colLines[j] == false) {countZeros++; colNo = j;}
        }
        // لو الصف فيه زيرو واحد و العمود ده متعملش عليه خط قبل كدا اعمل عليه خط
        if (countZeros == 1 && colLines[colNo] == false ){
            colLines[colNo] = true;
            solution[i] = colNo; // asigning person index to one job index from firstMatrix
        }
    }
    // check col zeros (Horizontal lines) (one zero in a col = horizontal line)
    for (int j = 0; j < n; j++) {
        int countZeros = 0;
        int rowNo = 0; // store row number to make horizontal line
        for (int i = 0; i < n; i++) {
            // هنا فيه حاجة زيادة لازم تتأكد ان الزيرو ده مش معمول عليه خط فيرتيكال اصلا
            if(M[i][j] == 0 && colLines[j] == false) {countZeros++; rowNo = i;}
        }

        if (countZeros == 1 && rowLines[rowNo] == false){
            rowLines[rowNo] = true;
            solution[rowNo] = j; // asigning person index to one job index from firstMatrix
        }
    }
    // Check last time that no zeros left uncovered
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if(M[i][j] == 0 && colLines[j] == false && rowLines[i] == false ) {
            rowLines[i] = true;
            solution[i] = j; // asigning person index to one job index from firstMatrix
            }
        }
    }
```

```java
            // Get number of all Lines (horizontal or vertical)
            int noLines = calcLines(rowLines,colLines);
                System.out.println("noLines => "+ noLines) ;

            // Check if (number of Lines = size of matrix)
            if (noLines == n) {
                return solution;
            } // return solution Done#

            else {
                // need one more step
                System.out.println("----------------Step 4: Create additional zeros----------------");
                // Step-4: Create additional zeros
                // Find the smallest uncovered number with no lines O(n**2)
                int minNumber = Integer.MAX_VALUE;
                for (int i = 0; i < n; i++) {
                    for (int j = 0; j < n; j++) {
                        if (colLines[j] == false && rowLines[i] == false && M[i][j] < minNumber) {minNumber = M[i][j];}
                    }
                }
                // Subtract this number from all uncovered elements
                for (int i = 0; i < n; i++) {
                    for (int j = 0; j < n; j++) {
                        if (colLines[j] == false && rowLines[i] == false) {M[i][j]-=minNumber;}
                    }

                }
                // Add it to all elements that are covered twice
                for (int i = 0; i < n; i++) {
                    for (int j = 0; j < n; j++) {
                        if (colLines[j] == true && rowLines[i] == true) {M[i][j]+=minNumber;}
                    }
                }
                displayMatrix(M);
                return drawZeroslines(M);
            }
        }
        /*---------------------------------------------------------------------------------------------------------------*/
```

3. Main Method

The main method executes the program and calling all methods to implement the Hungarian Algorithm steps, at first get two input's from the user: size of the balanced matrix and the matrix values then create a copy of this matrix and save it at (firstMatrix), after that it will start to execute algorithm steps to get the minimum optimal solution:

Step-1: Subtract Row minimum by calling subRowmin(matrix);

Step-2: Subtract Col minimum   by calling subColmin(matrix);

Step-3: Cover all zeros with a minimum number of lines horizontal or vertical by calling drawZeroslines(matrix,firstMatrix); and store it at solution array.

At the end it will display the solution array (ex. Each person assigns to each job to get the minimum optimal solution)   Person = i+1   job = solution[i]+1

And display the optimal solution from the (firstMatrix)

sumJobsvalues +=firstMatrix[i][solution[i]]

```java
public static void main(String[] args) {
    // get Size of the Balanced Matrix
    System.out.println("Enter Size for Balanced Matrix");
    int n = input.nextInt();
    // Create Balanced Matrix
    int matrix[][] = new int[n][n];
    // get matrix inputs
    System.out.println("Enter Elements for Balanced Matrix "+n+"X"+n);
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            matrix[i][j]=input.nextInt();
        }
    }
    // Save a copy of the Original Matrix (use it at the end to get solution)
    int firstMatrix[][] = new int[n][n];
    saveFirstmatrix(matrix,firstMatrix);
    System.out.println("----------------Step-1: Subtract Row minimum----------------");
    // Step-1: Subtract Row minimum
    subRowmin(matrix);
    displayMatrix(matrix);
    System.out.println("----------------Step-2: Subtract Col minimum----------------");
    // Step-2: Subtract Col minimum
    subColmin(matrix);
    displayMatrix(matrix);
    System.out.println("----------------Step 3: Cover all zeros with a minimum number of lines----------------");
    // Step-3: Cover all zeros with a minimum number of lines (horizontal or vertical)
    int solution [] = drawZeroslines(matrix); // index = person(row)  value = Job(col)
    System.out.println("----------------------------------------Solution----------------------------------------");
    //Display Solution
        int sumJobsvalues = 0;
        for (int i = 0; i < solution.length; i++) {
            int personCount = i+1;
            int jobCount = solution[i]+1;
            sumJobsvalues+=firstMatrix[i][solution[i]]; // رقم الاندكس لكل عنصر فيه هو رقم الصف والقيمة هيا رقم العمود
            System.out.println("Person "+ personCount +" "+"Assign to " +"Job "+jobCount);
        }
        System.out.println("The optimal value equals => "+ sumJobsvalues  );
}
}
```

2. Exhaustive Search (brute force)

2.1. Overview

The Brute Force Method that calculates the cost of all assignments is a very complex method, especially for large assignments. Suppose we take nxn cost matrix assignment. There are n option for the first assignment and $n-1$ option for the second assignment, respectively. Thus, the solution has an exponential run time. There are $n!$ assignments for n. This solution may be suitable for small n values, Table below shows the iterator numbers of the Brute Force method.

Table Size	Solution	Number of iterations
3x3	3!	6
4x4	4!	24
5x5	5!	120
6x6	6!	720
7x7	7!	5040
8x8	8!	40320
9x9	9!	362880
10x10	10!	3628800

2.2 Code description

```java
package exhaustivesearch;
 // @author Mina Romany
import java.util.Arrays;
import java.util.Scanner;
public class ExhaustiveSearch {
    static Scanner input = new Scanner(System.in);

    /*---------------------------------------- Helpful Methods ------------------------------------------*/
    // make array(n) from 1 to n thedn get all permutations for this array
    static int[] firstPermutearray (int arr[]) {
        int n = arr.length;
        for (int i = 0; i < n; i++) {
            arr[i] = i+1;
        }
        return arr;
    }

    // sum all values(jobs)from a matrix using array values as index
    static int sumvalues (int arr[], int M[][]) {
        int n = arr.length;
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum+=M[i][arr[i]-1];
        }
        return sum;
    }

    // Copy array elements in newArray
    static int[] copyArray (int arr[]) {
        int n = arr.length;
        int newArray[] = new int [n];
        for (int i = 0; i < n; i++) {
            newArray[i] = arr[i];
        }
        return newArray;
    }
    /*-----------------------------------------------------------------------------------------------------*/
```

```java
/*------------------------------------- Main Method -------------------------------------------------------*/
public static void main(String[] args) {
    // get Size of the Balanced Matrix
    System.out.println("Enter Size for Balanced Matrix");
    int n = input.nextInt();
    // Create Balanced Matrix
    int matrix[][] = new int [n][n];
    // get matrix inputs
    System.out.println("Enter Elements for Balanced Matrix "+n+"X"+n);
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            matrix[i][j]=input.nextInt();
        }
    }


    // Assignment Problem (fisrt permutation)
    int A[] = new int [n];
    A = firstPermutearray(A); // Create array save all possibilities  ex. n =4  A = {1,2,3,4} to get permutations

    int solution[] = new int [n]; // for saving array (jobs indexes) that get the minimum optimal solution
    solution = A; // Save first permutation jobs indexe

    int minOptimal = sumvalues(A,matrix); // for saving the minimum optimal solution
```

```java
        // heap's algorithm, iterative --to get all permutations
        // make idx array with zeros
        int[] idx = new int[A.length];
        Arrays.fill(idx, 0);

        for (int i = 1; i < A.length;) {
          if (idx[i] < i) {
            int swap = i % 2 * idx[i];
            int tmp = A[swap];
            A[swap] = A[i];
            A[i] = tmp;

            // Assignment Problem (new permutation)
            int sum = 0;
            sum = sumvalues(A,matrix);
            if (sum < minOptimal) {
                minOptimal = sum;  // new minimum Optimal solution
                solution = copyArray(A); // Save jobs indexe in solution array
            }

            idx[i]++;
            i = 1;
          } else {
            idx[i++] = 0;
          }
        }


        System.out.println("-------------------------------------Solution-------------------------------------");
        //Display Solution
        for (int i = 0; i < solution.length; i++) {
            int personCount = i+1;
            System.out.println("Person "+ personCount +" "+"Assign to " +"Job "+solution[i]);
        }
        System.out.println("The optimal value equals => "+ minOptimal);

    }
}
```