Hungarian Algorithm Implementation for The Assignment Problem

By Hiba Azhari

Report

The Assignment Problem

In this problem, a set of jobs, operators and cost for each operator to execute each job, are given. The objective is to assign the jobs with minimum total cost (maximum total cost in maximization variation). A popular method for solving this problem is the Hungarian Algorithm developed in 1955 by Harold Kuhn[1]. The method consists of five main steps which are:

1. Row reduction
2. Column reduction
3. Crossing of zeros
4. Subtraction/addition of minimum uncrossed value
5. Assignment

In the program, the job-to-operator assignment costs are represented in a 2D integer array (Fig.1)

Fig. 11

The five steps are implemented in functions applied to the array to solve the problem. The functions are outlined in the next section along with more data structures needed for implementation.

Code Overview

The program consists of an *AssignmentProblem* class and a *UI* class. The *UI* class is simply a collection of functions used to provide user interaction and also contains the main function. This report focuses on the functionality on the *AssignmentProblem* class side.

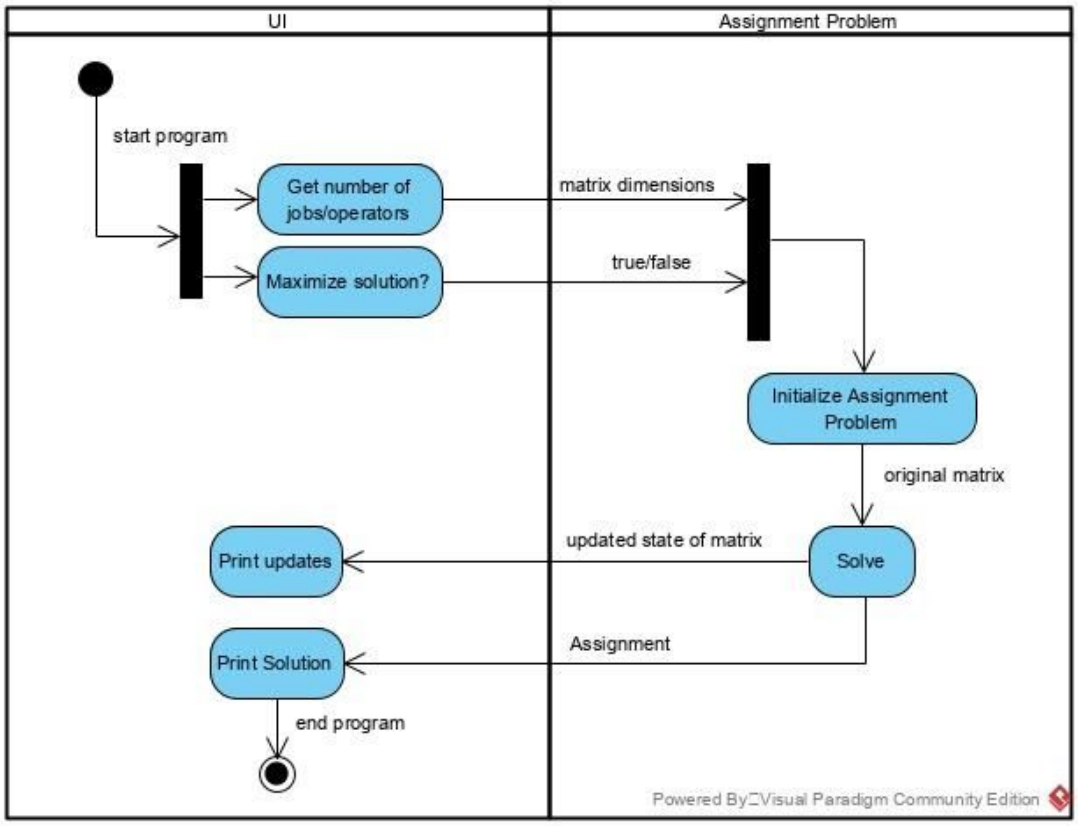

Fig. 12

Data Structures

The *AssignmentProblem* class has 5 main data structures used in solving the problem. These are:

- *originalMatrix*: int[][] to store assignment costs
- *reducedMatrix*: int[][] to store costs after reducing and doing calculations
- *crossedCols*: HashSet<Integer> to store indices of columns crossed out in step 3
- *crossedRows*: HashSet<Integer> to store indices of rows crossed out in step 3
- *assignment*: HashMap<Integer,Integer> to map jobs to their assigned operator

Other attributes

- *maximize*: boolean to indicate whether the aim is to maximize or minimize score
- *maxValue*: int stores the maximum value in the matrix for maximization problem
- *score*: int stores the total cost calculated after assignment
- *assignments*: int[][] stores assignments as [row index, column index]
- *costs*: int[] stores costs of assignments chosen

Functions

The program consists of 9 functions:

- *reduceRows()* [Step 1]
- *reduceCols()* [Step 2]
- *crossOutZeros()*: [Step 3]
- *addSubLeastUncrossed()*: [Step 4]
- *assignmentPossible(int i)*: Checks whether an assignment is possible in the matrix
- *calcAssignment()* [Step 5]
- *solve()*: Calls the steps in the required order
- *getMaxValue()*: find max cost in matrix (used when maximizing cost)
- *Maximize()*: subtracts all costs from max cost in matrix (used when maximizing cost)

The functions are documented in depth below:

reduceRows()

- **Description**: Function loops through each row in *originalMatrix* twice. Once to find the minimum value and again to subtract it from all values.
- **Analysis**:
    - Time: Since the entire matrix (N-jobs x N-operators) is traversed twice, the time complexity is $O(2N^2) = O(N^2)$ complexity
    - Space: one variable "least" is used regardless of matrix size: O(1)

reduceCols()

- **Description**: As in reduceRows but through columns
- **Analysis**: (as in rows)
    - Time: $O(N^2)$
    - Space: O(1)

crossOutZeros()

- **Description**: The function "crosses out" a row or column by adding its index to the corresponding HashSet *crossedRows* or *crossedCols*. The algorithm works by scanning rows first. If a zero exists in the row and it is the only zero in the row, its corresponding column is crossed out. The same is true for columns. If a zero exists in a column and it is the only zero in its column, the corresponding row is crossed out.

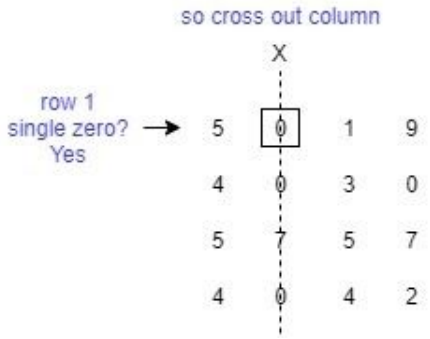

Fig. 13

- **Analysis**:
  - Time: The crossing-out procedure is repeated twice, once for rows and again for columns. Each time, all the rows/cols are traversed once giving a complexity of $O(2N^2) \Rightarrow O(N^2)$
  - Space: Two HashSets are used to store the crossed out rows and cols: $O(2N) \Rightarrow O(N)$
- **Pseudocode**:

```
for each row
  set zero pos to -1
  onezero = true
    for each column
      if column is crossed continue
      if cost == 0
        if pos < 0 {
            no previous zero has been found, set pos = current col index
        }
        else set onezero to false and break
  if pos >= 0 (zero found) and onezero add pos to crossedCols

*repeat for columns*
```

addSubLeastUncrossed()

- **Description:** Function loops through the entire array and records the minimum cost value *least*. It then loops through again. If an entry's row and column index are both crossed (row in crossedRows and column in crossedCols) it adds *least* to it. If neither the entry's row index nor column index is present in the corresponding *crossed* set, it subtracts *least* from it.
- **Analysis**:
  - Time: The matrix is first traversed to find the minimum uncovered element then again to do adding and subtracting: $O(N^2)$. Checking whether a row/col is crossed is O(1) since HashSets are used leading to $O(N^2)$ overall complexity.
  - Space: one variable "least" is used regardless of matrix size: O(1)

- **Pseudocode**:

```
set least = Integer.MAX_VALUE
for each row
  if the row's index is not in crossedRows
  for each column
    if column index is not in crossedCols
      set least to entry value if entry < least
for each row
  for each column
    if row in crossedRows and col in crossedCols
      entry += least
    if row not in crossedRows and col not in crossedCols
      entry -= least
```

assignmentPossible(int i)

- **Description:** Function attempts, recursively, to find an assignment in the matrix through backtracking. When the function 'attempts' an assignment, it adds the column as key and row as value to the *assignment* HashMap. This allows quick searching for whether a column is taken.
- **Analysis**:
  - Time: Since the algorithm attempts an assignment only when the entry value is '0', the worst case would be when all rows are filled with zeros except the last.

0	0	0	0
0	0	0	0
0	0	0	0
1	2	3	4

    In which the algorithm will attempt all possible permutations leading to a complexity of O(N!). However, such a case is highly unlikely.
  - Space: A map with N entries is used to store the assignments therefore: O(N)
- **Pseudocode**:

```
if i is last row in matrix return true

in row i, for each col j:
  if the entry == 0 and !assignments
    try assigning the column to the row
    attempt assignment for remaining rows(call assignmentPossible(i+1))
    if possible return true
    if not possible
      remove assignment of i to j

if loop finished return false
```

calcAssignment()

- **Description**: After an assignment is found, this function sets up *score, assignments* and *costs* to be printed in solution. For each column in the matrix, the index and corresponding row index (from *assignment* map) are added to the *assignments* array. The cost is retrieved from the *originalMatrix* and stored in *costs*. The total cost is added up and stored in *score*.
- **Analysis**:
    - Time: The function loops through the rows in O(N), retrieves the assignment O(1), and updates *costs* and *score* O(1). Overall complexity: O(N).
    - Space: Only additional space used is for output: O(1)

findMaxValue()

- **Description**: Loops through the array to find the maximum cost and store it in *maxValue*.
- **Analysis**:
    - Time: Single loop through matrix: $O(N^2)$
    - Space: One variable *maxValue* is used: O(1)

maximize()

- **Description**: Loops through the array to subtract each value from *maxValue* to convert the matrix to a maximization problem. It is called again after an assignment is found to return the matrix to its original state and retrieve costs.
- **Analysis**:
    - Time: Single loop through matrix: $O(N^2)$
    - Space: No additional space used: O(1)

solve()

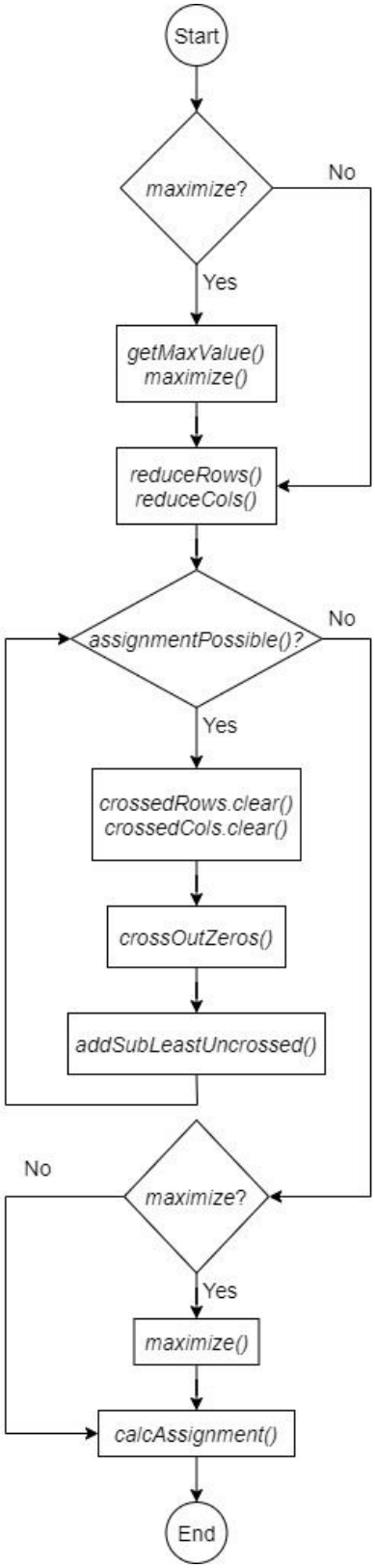

Fig.14: solve() function flowchart

The solve function incorporates all the functions in the correct flow, as shown in Fig.1, to execute the algorithm.

Sample Run

This section is a demonstration of a sample 4x4 assignment problem solved by the program. First the user must enter the matrix information:

```
--- Assignment Problem Simulation ---

Please enter number of jobs/operators (1 - 30): 4

          Operators
          1  2  3  4
       1  #  #  #  #
       2  #  #  #  #
 Jobs  3  #  #  #  #
       4  #  #  #  #

Please enter job costs for operator 1

Job 1 cost (OP 1):
```

Then the matrix is initialized and the user can choose to either maximize or minimize total score:

```
Please enter job costs for operator 4

Job 1 cost (OP 4):1
Job 2 cost (OP 4):3
Job 3 cost (OP 4):13
Job 4 cost (OP 4):15

           Operators
           1  2  3  4
        1  4  8  2  1
        2  5  7  9  3
 Jobs   3  2  4  5 13
        4  1  6  4 15

Maximize(M) or Minimize(m) score?
```

In the case of maximizing, the matrix is preprocessed:

```
Maximize(M) or Minimize(m) score?M

To maximize, all numbers must be subtracted from the largest number in the matrix:


           Operators
           1  2  3  4
        1 11  7 13 14
        2 10  8  6 12
 Jobs   3 13 11 10  2
        4 14  9 11  0
```

Then the algorithm proceeds:

```
First the rows are reduced:

           Operators
              1  2  3  4
          1   4  0  6  7
          2   4  2  0  6
 Jobs     3  11  9  8  0
          4  14  9 11  0


Then columns are reduced:

           Operators
              1  2  3  4
          1   0  0  6  7
          2   0  2  0  6
 Jobs     3   7  9  8  0
          4  10  9 11  0
```

```
If no assignment is possible, zeros are crossed out with minimum number of lines:

           Operators
              1  2  3  4
          1   0  0  6  7  X
          2   0  2  0  6  X
 Jobs     3   7  9  8  0
          4  10  9 11  0
                      X


Then the smallest uncrossed value is found, subtracted from all other uncrossed
 values and added to values at intersections:

           Operators
              1  2  3  4
          1   0  0  6 14
          2   0  2  0 13
 Jobs     3   0  2  1  0
          4   3  2  4  0
```

```
An assignment is now possible. The result is:


           Operators
              1  2  3  4
        1  0 *0  6 14
        2  0  2 *0 13
 Jobs   3 *0  2  1  0
        4  3  2  4 *0


           Operators
              1  2  3  4
        1  4  8  2  1
        2  5  7  9  3
 Jobs   3  2  4  5 13
        4  1  6  4 15

Operator: 1 => Job: 3
Operator: 2 => Job: 1
Operator: 3 => Job: 2
Operator: 4 => Job: 4

Total Cost = 2 + 8 + 9 + 15 = 34
```