Assignment Problems: Hungarian Method

Yukta Salunkhe

May 3, 2023

Contents

Overview

Given 'n' resources, 'n' jobs and the efficiency of each resource for each job, the problem is to assign each resource to one and only one job in such a way that the measure of efficiency/effectiveness is optimised (Maximised or Minimised)

Major Objectives:

- Minimize Cost
- Maximize Profit

Assignment Problems

Characteristics:

- Number of Jobs = Number of Resources (in case of unbalanced assignments, Dummy is used)
- One job can be assigned to only one resource.
- One resource can do only one job.

Assignment Problems

$$X_{ij} = \begin{cases} 1, & \text{if ith person is assigned jth job} \\ 0, & \text{if not} \end{cases}$$

C_{ij} = the cost of person i performing job j(1)

Objective

Minimize $\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$

Conditions:

$\sum_{i=1}^{n} x_{ij} = 1$, 1 person can do at most 1 job

$\sum_{j=1}^{n} x_{ij} = 1$, 1 job can be assigned to atmost 1 person

$x_{ij} >= 0$

Hungarian Method

The Hungarian methodis a combinatorial optimization algorithm that solves the assignment problem in polynomial time $O(n^3)$.

Hungarian Method

Consider a minimization problem:

$$\begin{pmatrix} 1 & 4 & 5 \\ 5 & 7 & 6 \\ 5 & 8 & 8 \end{pmatrix} \iff$$

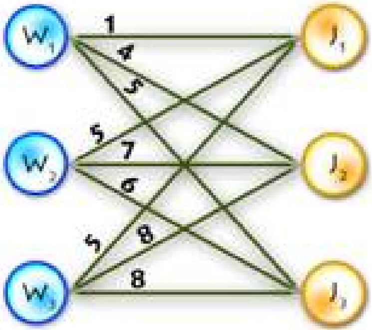

Figure 1: Matrix Representation and Graphical Representation

Hungarian Method

Consider above representation for problem where we have to assign 3 workers(rows) to 3 Jobs(columns),given their cost.

Objective:

- From Matrix Representation: Find an assignment for each job, such that the cost of completing all works, by the workers is minimum.
- From Graphical Representation: We have a complete bipartite graph $G = (S , T ; E)$ G=(S, T; E) with n worker vertices (S) and n job vertices (T), and the edges (E) each have a nonnegative cost c(i,j). We want to find a perfect matching with a minimum total cost.

Hungarian Method

Theorem

If a number is added to or subtracted from all of the entries of any one row or column of a cost matrix, then an optimal assignment for the resulting cost matrix is also an optimal assignment for the original cost matrix.

Optimality Test

Minimum number of covering lines should be equal to dimension of rows(/columns).

Algorithm

Example 1: A department head has 3 subordinates and 3 tasks have to be performed. Subordinates differ in efficiency and tasks differ in their difficulty.Efficiency matrix below, defines time taken for each person to complete each task. How the tasks should be allocated to so as to minimize the total hours?

	Job 1	Job 2	Job 3
W 1	10	16	32
W 2	14	22	40
W 3	22	24	34

Algorithm

Step1: For each row of the matrix, find the smallest element and subtract it from every element in its row.

	Job 1	Job 2	Job 3
W 1	10	16	32
W 2	14	22	40
W 3	22	24	34

	Job 1	Job 2	Job 3
W 1	0	6	22
W 2	0	8	26
W 3	0	2	12

Figure 3: Step 1

Algorithm

Step2: Repeat the same for all columns.

	Job 1	Job 2	Job 3
W 1	0	6	22
W 2	0	8	26
W 3	0	2	12

	Job 1	Job 2	Job 3
W 1	0	4	10
W 2	0	6	14
W 3	0	0	0

Figure 4: Step 2

Algorithm

Step3: Cover all zeros in the matrix using minimum number of horizontal and vertical lines.

	Job 1	Job 2	Job 3
W 1	0	4	10
W 2	0	6	14
W 3	0	0	0

Figure 5: Step 3

Algorithm

Step4: Test for Optimality: If the minimum number of covering lines is n, an optimal assignment is possible and we are finished. Else if lines are lesser than n, we haven't found the optimal assignment, and must proceed to step 5.

Assignment Matrix formed is NOT OPTIMAL.
So we proceed to step 5

Algorithm

Step5: Determine the smallest element in the matrix, not covered by N lines.Subtract this minimum element from all uncovered elements and add the same element at the intersection of horizontal and vertical lines. Thus the second modified matrix is obtained. Now repeat step 3,step 4, till optimality test gets satisfied.

	Job 1	Job 2	Job 3
W 1	0	4	10
W 2	0	6	14
W 3	0	0	0

$\Rightarrow$

	Job 1	Job 2	Job 3
W 1	0	0	6
W 2	0	2	10
W 3	4	0	0

Algorithm

Final modified matrix:

	Job 1	Job 2	Job 3
W 1	10	16	32
W 2	14	22	40
W 3	22	24	34

Thus job assignments would be:
W1 $\rightarrow J2, W2 \rightarrow J1, W3 \rightarrow J3$
So, minimum hours = 14+16+34 = 64

Example

Example 2: A car hire company has one car at each of 4 depots I, II, III, and IV.A customer requires a car in each town namely A, B, C, and D. Distance between depots (origins) and towns (destinations) are given in the following distance matrix. Assign the cars to the towns, minimizing distance needed to travel.

Example

Distance Matrix:

	I	II	III	IV
A	10	12	19	11
B	5	10	7	8
C	12	14	13	11
D	8	15	11	9

Figure 7: Example 2

Lets solve this using python code $\rightarrow$

Code

```python
import numpy as np

def min_zero_row(zero_mat, mark_zero):

    '''
    The function can be splitted into two steps:
    #1 The function is used to find the row which containing the fewest 0.
    #2 Select the zero number on the row, and then marked the element corresponding row and column as False
    '''

    #Find the row
    min_row = [99999, -1]

    for row_num in range(zero_mat.shape[0]):
        if np.sum(zero_mat[row_num] == True) > 0 and min_row[0] > np.sum(zero_mat[row_num] == True):
            min_row = [np.sum(zero_mat[row_num] == True), row_num]

    # Marked the specific row and column as False
    zero_index = np.where(zero_mat[min_row[1]] == True)[0][0]
    mark_zero.append((min_row[1], zero_index))
    zero_mat[min_row[1], :] = False
    zero_mat[:, zero_index] = False

def mark_matrix(mat):

    '''
    Finding the returning possible solutions for LAP problem.
    '''

    #Transform the matrix to boolean matrix(0 = True, others = False)
    cur_mat = mat
    zero_bool_mat = (cur_mat == 0)
    zero_bool_mat_copy = zero_bool_mat.copy()

    #Recording possible answer positions by marked_zero
    marked_zero = []
    while (True in zero_bool_mat_copy):
        min_zero_row(zero_bool_mat_copy, marked_zero)
```

Code

```python
    while (True in zero_bool_mat_copy):
        min_zero_row(zero_bool_mat_copy, marked_zero)

    #Recording the row and column positions seperately.
    marked_zero_row = []
    marked_zero_col = []
    for i in range(len(marked_zero)):
        marked_zero_row.append(marked_zero[i][0])
        marked_zero_col.append(marked_zero[i][1])

    #Step 2-2-1
    non_marked_row = list(set(range(cur_mat.shape[0])) - set(marked_zero_row))

    marked_cols = []
    check_switch = True
    while check_switch:
        check_switch = False
        for i in range(len(non_marked_row)):
            row_array = zero_bool_mat[non_marked_row[i], :]
            for j in range(row_array.shape[0]):
                #Step 2-2-2
                if row_array[j] == True and j not in marked_cols:
                    #Step 2-2-3
                    marked_cols.append(j)
                    check_switch = True

        for row_num, col_num in marked_zero:
            #Step 2-2-4
            if row_num not in non_marked_row and col_num in marked_cols:
                #Step 2-2-5
                non_marked_row.append(row_num)
                check_switch = True
    #Step 2-2-6
    marked_rows = list(set(range(mat.shape[0])) - set(non_marked_row))

    return(marked_zero, marked_rows, marked_cols)

def adjust_matrix(mat, cover_rows, cover_cols):
    cur_mat = mat
```

Code

```python
def adjust_matrix(mat, cover_rows, cover_cols):
    cur_mat = mat
    non_zero_element = []

    #Step 4-1
    for row in range(len(cur_mat)):
        if row not in cover_rows:
            for i in range(len(cur_mat[row])):
                if i not in cover_cols:
                    non_zero_element.append(cur_mat[row][i])
    min_num = min(non_zero_element)

    #Step 4-2
    for row in range(len(cur_mat)):
        if row not in cover_rows:
            for i in range(len(cur_mat[row])):
                if i not in cover_cols:
                    cur_mat[row, i] = cur_mat[row, i] - min_num
    #Step 4-3
    for row in range(len(cover_rows)):
        for col in range(len(cover_cols)):
            cur_mat[cover_rows[row], cover_cols[col]] = cur_mat[cover_rows[row], cover_cols[col]] + min_num
    return cur_mat

def hungarian_algorithm(mat):
    dim = mat.shape[0]
    cur_mat = mat

    #Step 1 - Every column and every row subtract its internal minimum
    for row_num in range(mat.shape[0]):
        cur_mat[row_num] = cur_mat[row_num] - np.min(cur_mat[row_num])

    for col_num in range(mat.shape[1]):
        cur_mat[:,col_num] = cur_mat[:,col_num] - np.min(cur_mat[:,col_num])
    zero_count = 0
    while zero_count < dim:
        #Step 2 & 3
        ans_pos, marked_rows, marked_cols = mark_matrix(cur_mat)
```

Code

```python
    while zero_count < dim:
        #Step 2 & 3
        ans_pos, marked_rows, marked_cols = mark_matrix(cur_mat)
        zero_count = len(marked_rows) + len(marked_cols)

        if zero_count < dim:
            cur_mat = adjust_matrix(cur_mat, marked_rows, marked_cols)

    return ans_pos

def ans_calculation(mat, pos):
    total = 0
    ans_mat = np.zeros((mat.shape[0], mat.shape[1]))
    for i in range(len(pos)):
        total += mat[pos[i][0], pos[i][1]]
        ans_mat[pos[i][0], pos[i][1]] = mat[pos[i][0], pos[i][1]]
    return total, ans_mat

def main():

    '''Hungarian Algorithm:
    Finding the minimum value in linear assignment problem.
    Therefore, we can find the minimum value set in net matrix
    by using Hungarian Algorithm.'''
    #The matrix who you want to find the minimum sum
    cost_matrix = np.array([[10, 12, 19, 11],
                [5, 10, 7, 8],
                [12, 14, 13, 11],
                [8, 15, 11, 9]])
    ans_pos = hungarian_algorithm(cost_matrix.copy())#Get the element position.
    ans, ans_mat = ans_calculation(cost_matrix, ans_pos)#Get the minimum or maximum value and corresponding matrix.

    #Show the result
    print(f"Linear Assignment problem result: {ans:.0f}\n")
    for i in range(ans_mat[0].size):
        for j in range(ans_mat[0].size):
            if(ans_mat[i][j] != 0):
```

Code

```python
def main():

    '''Hungarian Algorithm:
    Finding the minimum value in linear assignment problem.
    Therefore, we can find the minimum value set in net matrix
    by using Hungarian Algorithm.'''
    #The matrix who you want to find the minimum sum
    cost_matrix = np.array([[10, 12, 19, 11],
                            [5, 10, 7, 8],
                            [12, 14, 13, 11],
                            [8, 15, 11, 9]])
    ans_pos = hungarian_algorithm(cost_matrix.copy())#Get the element position.
    ans, ans_mat = ans_calculation(cost_matrix, ans_pos)#Get the minimum or maximum value and corresponding matrix.

    #Show the result
    print(f"Linear Assignment problem result: {ans:.0f}\n")
    for i in range(ans_mat[0].size):
        for j in range(ans_mat[0].size):
            if(ans_mat[i][j] != 0):

                print(f"Car{i+1} -> Town {j+1}, Distance: {ans_mat[i][j]}")
    print(f"Minimum Distance covered: {ans:.0f}\n")

if __name__ == '__main__':
    main()
```

✓ 0.3s

```
Linear Assignment problem result: 38

Car1 -> Town 2, Distance: 12.0
Car2 -> Town 3, Distance: 7.0
Car3 -> Town 4, Distance: 11.0
```

Example

Solution:

Car1 $\rightarrow$ $Town2, Distance : 12.0$

Car2 $\rightarrow$ $Town3, Distance : 7.0$

Car3 $\rightarrow$ $Town4, Distance : 11.0$

Car4 $\rightarrow$ $Town1, Distance : 8.0$

Minimum Distance covered: 38

Code

Time Complexity: $O(n^3)$
Space complexity : $O(n^2)$

Bipartite Graph and Hungarian Method

Another Approach:

Augmenting Path:
Path in the bipartite graph that starts and ends at unmatched vertices and alternates between matched and unmatched vertices.

The Hungarian algorithm starts with an initial matching and it iteratively searches for augmenting paths until no more augmenting paths can be found. Once no more augmenting paths can be found, the matching found is guaranteed to be a maximum matching. It thus tries to find path where cost is minimum, and matching is maximum.

- Balanced Problems (Covered in the slides)
- Unbalanced Problems (Add Dummy to make it balanced and solve in the same way)
- Profit Maximization Problems
    - Subtract all values from the maximum value in the matrix.
    - Then solve it similar to the Minimization problem, on the matrix (Regret Matrix) formed.
    - (Maximizing the profit == Minimizing the difference between the cell value and maximum value)

Applications

- Minimize the cost/time required to complete tasks.
- Maximize the profit by assigning best job to best person.
- Assign jobs to machines.
- Assign vehicles to routes.
- Assign sales representative to sales territories.
- Minimize time of arrival and departure of airlines.

References

- https://en.wikipedia.org/wiki/Hungarian_algorithm
- https://www.uobabylon.edu.iq/eprints/publication_2_24362_31.pdf
- https://python.plainenglish.io/hungarian-algorithm-introduction-python-implementatio
- https://www.geeksforgeeks.org/hungarian-algorithm-assignment-problem-set-1-introduct

Thank You!