Industrial Engineering Journal
ISSN: 0970-2555
Volume : 52, Issue 11, No. 3, November : 2023

OPTIMIZATION OF TRANSPORTATION COST THROUGH DATA SCIENCE USING PYTHON

M. Lakshmana Rao, Department of Mechanical Engg Madanapalle Institute of Technology & Science, Madanapalle, A.P., India
B. Raghukumar, Dept of Mechanical Engg PVP Siddhartha Institute of Technology, Kanur, Vijayawada
G. Harinath Gowd, Dept of Mechanical Engg ,Vemu Institute of Technology, Tirupati
N. Gangi Setty, Department of Management studies, Madanapalle Institute of Technology & Science, Madanapalle, A.P., India

Abstract
Operations Research seeks the optimal solution to a problem. This optimal solution is not just a solution that provides the best result, but the solutions have been calculated after considering the various aspects of time and cost constraints. Linear Programming is the analysis of problems in which a linear function of a number of variables is to be optimized (maximized or minimized) when those variables are subject to a number of constraints in the mathematical linear inequalities. The transportation Model is one type of linear programming which is to transport similar quantities which are initially stored at various origins(supply) to different destinations (demand) in such a way that the total transportation cost is minimum. In this modern automated world still, companies are facing a high amount to transport their goods all over the country, which impacts fits. So, we decided to use the Linear Programming algorithm with Python. For getting the optimal solution used linear Programming optimal algorithm, which helps the industries to optimize their transportation cost. This study helpful for developing an algorithm for logistic industry.

1. Introduction

Operations Research provides a facility to decision maker to evaluate the given problems, identify the alternative solutions, recognize the constraints and then assist the decision maker to have the best possible solution available, which is known as optimal solution. Operation Research also provides the quantitative and qualitative aid to the problems, so that it will become easier for decision maker to predict the future outcomes of the solution. The uncertainty of future and complexities of present scenarios increases the responsibility of decision maker to take the accurate decision for the organization. Operations Research theory makes the problems of real-life more structured and hence, easily solvable and has correct answers. In Operations Research, Linear Programming is one of the models in mathematical programming, which is very broad and vast. Mathematical programming includes many more optimization models known as Stochastic programming, Integer Programming, and Dynamic Programming – each one of them is an efficient optimization technique to solve the problem with a specific structure, which depends on the assumptions made in formulating the model. We can remember that the general linear programming model is based on assumptions [1]. A few of the areas where operations research finds application are presented in Fig.1.

Industrial Engineering Journal
ISSN: 0970-2555
Volume : 52, Issue 11, No. 3, November : 2023

Figure 1. Applications of operations research.

2. Methodology

The North West Corner (NWC) rule is used to find the initial feasible solution to the considered transportation problem as mentioned in Table 1. The obtained solution is optimized using MODI method in the next stage. A python code was used to obtain the solutions of NWC and MODI methods.

Table 1. Transportation problem considered in the present work [2].

To / Source	A	B	C	Supply
X	5 (20)	6 (30)	4	50
Y	6	6 (40)	3	40
Z	3	9 (25)	6 (35)	60
Demand	20	95	35	150

2.1 North West Corner (NWC) Algorithm

The NWC algorithm is one of the methods to obtain a basic feasible solution of transportation problems [3,4]. The steps involved in NWC method are as follows:

- **Step 1:** Formulate the transportation model into a 2-D matrix along with supply and demand requirements.
- **Step 2:** Now check the matrix for balanced. If it is balanced move on with the below steps or add the particular row or column with the particular element in the supply or demand.
- **Step 3:** Choose the element based on direction North-West and allocate the value which is smaller between supply and demand.
- **Step 4:** Now we have to delete the smaller from the bigger number and delete that row or column which is taken into consideration in step 2.
- **Step 5:** Now after the reduction consider the matrix again consider North-West direction and repeat the steps 2 and 3 until the values of supply and demand becomes zero.
- **Step 6:** Obtain the initial basic feasible solution.

2.2 MODI method with algorithm

The obtained initial feasible solution is optimized using MODI method [5]. The steps involved in MODI method are described below.

- **Step 1:** After getting initial feasible solution using North West Corner Method we have to check for optimal solution using MODI method.
- **Step 2:** Finding the values of two variable's u_i and v_j with $u_i + v_j = c_{ij}$
- **Step 3:** Finding the opportunity cost using $c_{ij} - (u_i + v_j)$.
- **Step 4:** We have to check the sign for the opportunity if positive or zero the given solution is the optimal solution. If any of the unoccupied cell opportunity is negative then futher savings can be possible in transportation.

Industrial Engineering Journal
ISSN: 0970-2555
Volume : 52, Issue 11, No. 3, November : 2023

- **Step 5:** Select the least possible opportunity cell in the matrix which is helpful in the next step.
- **Step 6:** ow we have to draw a closed loop with right angle turn with only occupied cells and the previous selected cell.
- **Step 7:** Give the plus or minus to the closed loop starting with the selected cell with positive sign.
- **Step 8**: Now we have to make the unoccupied cell by considering the smallest value in the negative sign and add the value to the positive signs and subtract the value for negative signs. Now we unoccupied cell becomes an occupied cell.
- **Step 9:** Now we again do the opportunities if all are positive, we get the optimal solution to the problem else repeat the steps (2-8).

3. Optimization of transportation problem using python

The programming language used in the present study is Python. Codes were written on IDLE platform to estimate the initial feasible solution using NWC rule and subsequently optimize it based on MODI method. The code is described below.

```
import sys
def getCostMatrix():
    numRows = int(input('Enter the number of sources : '))+1
    numCols = int(input('Enter the number of destinations : '))+1
    costMatrix = []
    for i in range(numRows-1):
        rowCostArray=list(map(int, input('Enter the costs for source %s and the total supply at the end, separated by space\n'%(i+1)).split()))
        costMatrix.append(rowCostArray)
    rowCostArray = list(map(int, input('Enter the demand values for each destination separated by space\n').split()))
    costMatrix.append(rowCostArray)
    if len(costMatrix[numRows-1]) != numCols:
        costMatrix[numRows-1].append(0)
    return costMatrix

def printMatrix(matrixType, matrix):
    print("-------------------------------------------------------------------")
    if matrixType=='cost':
        print("Cost Matrix")
    elif matrixType=='allocation':
        print("Allocation Matrix")
    for i in range(len(matrix[0])-1):
        print('\tD%s'%(i+1), end='')
    print('\tSupply')

    for i in range(len(matrix)-1):
        print('S%s'%(i+1), end='')
        for j in range(len(matrix[0])):
            print('\t%s'%(matrix[i][j]), end='')
        print()
    print('Demand', end='')
    for i in range(len(matrix[0])):
        print('\t%s'%(matrix[-1][i]),end='')
    print("\n-------------------------------------------------------------------")
    print()

def isBalanced(costMatrix):
    return sum(costMatrix[-1])==sum([costMatrix[i][-1] for i in range(len(costMatrix))])

def getTotalCost(costMatrix):
    m = len(costMatrix)
    n = len(costMatrix[0])
    allocMatrix = [[0 for _ in range(len(costMatrix[0]))] for _ in range(len(costMatrix))]
    numAllocated = 0
    totalCost = 0
    i=0
    j=0
    while i<m-1 and j<n-1:
        x = min(costMatrix[i][n-1], costMatrix[m-1][j])
        costMatrix[m-1][j] -= x
        costMatrix[i][n-1] -= x
        numAllocated += 1

        allocMatrix[i][j] = x
```

```
        allocMatrix[m-1][j] = costMatrix[m-1][j]
        allocMatrix[i][n-1] = costMatrix[i][n-1]

        totalCost = totalCost + x*costMatrix[i][j]

        if costMatrix[m-1][j] < costMatrix[i][n-1]:
            j+=1
        elif costMatrix[m-1][j] > costMatrix[i][n-1]:
            i+=1
        else:
            i+=1
            j+=1

    return totalCost, numAllocated, allocMatrix

def isDegenerate(costMatrix, numAllocated):
    m = len(costMatrix)-1
    n = len(costMatrix[0])-1
    return numAllocated!=(m+n-1)

def balanceProblem(costMatrix):
    totalDemand = sum(costMatrix[-1])
    totalSupply = sum([x[-1] for x in costMatrix])
    if totalDemand > totalSupply:
        #add new row
        dummySource = [0 for _ in range(len(costMatrix[0]))]
        dummySource[-1] = totalDemand-totalSupply
        costMatrix.insert(-1, dummySource)
    else:
        for cost in costMatrix:
            cost.insert(-1, 0)
        costMatrix[-1].insert(-1, totalSupply-totalDemand)
        pass
    return costMatrix

def isIndependentAllocation( allocMatrix ):
    elimRows = [0 for _ in range(len(allocMatrix))]
    elimCols = [0 for _ in range(len(allocMatrix[0]))]
    while 1:
        flag = 0
        #eliminate row
        for i in range(len(allocMatrix)):
            if elimRows[i]==0:
                if len([allocMatrix[i][j] for j in range(len(allocMatrix[0])) if (elimCols[j]==0 and (allocMatrix[i][j]!=0 and allocMatrix[i][j]!=-1))]) < 2:
                    elimRows[i]=1
                    flag=1

        #eliminate column
        for j in range(len(allocMatrix[0])):
            if elimCols[j]==0:
                if len([allocMatrix[i][j] for i in range(len(allocMatrix)) if (elimRows[i]==0 and allocMatrix[i][j]!=0 and allocMatrix[i][j]!=-1) ]) < 2:
                    elimCols[j]=1
                    flag=1

        if flag==0:
            #either all cells are eliminated ---> independent allocation
            if 0 not in elimRows and 0 not in elimCols:
                return True,1,1
            else:
                #dependent allocation
                return False,elimRows,elimCols

def findUV( costMatrix, allocMat ):
    #find the max allocated row/col
    u = [None for _ in range(len(allocMat))]
    v = [None for _ in range(len(allocMat[0]))]

    maxRow=[-1,0] #(row no., allocs)
    for i in range(len(allocMat)):
        allocs = len([allocMat[i][j] for j in range(len(allocMat[0])) if allocMat[i][j]!=0])
        if allocs > maxRow[1]:
            maxRow[0] = i
            maxRow[1] = allocs

    maxCol = [-1,0]
    for j in range(len(allocMat[0])):
```

```
        allocs = len([ allocMat[i][j] for i in range(len(allocMat)) if allocMat[i][j]!=0 ])
        if allocs > maxCol[1]:
            maxCol[0] = j
            maxCol[1] = allocs

    if maxRow[1] > maxCol[1] :
        u[maxRow[0]] = 0
        for j in range(len(v)):
            if allocMat[maxRow[0]][j]!=0 and v[j] is None:
                v[j] = costMatrix[maxRow[0]][j] - u[maxRow[0]]

        for i in range(len(u)):
            for j in range(len(v)):
                if allocMat[i][j]!=0 and v[j] is not None and u[i] is None:
                    u[i] = costMatrix[i][j] - v[j]

    else:
        v[maxCol[0]] = 0
        for i in range(len(u)):
            if allocMat[i][maxCol[0]]!=0 and u[i] is None:
                u[i] = costMatrix[i][maxCol[0]] - v[maxCol[0]]

        for j in range(len(v)):
            for i in range(len(u)):
                if allocMat[i][j]!=0 and v[j] is None and u[i] is not None:
                    v[j] = costMatrix[i][j] - u[i]

    while None in u or None in v:
        if None in u:
            ind = u.index(None)
            for j in range(len(v)):
                if allocMat[ind][j]!=0 and v[j] is not None:
                    u[ind] = costMatrix[ind][j] - v[j]
        if None in v:
            ind = v.index(None)
            for i in range(len(u)):
                if allocMat[i][ind]!=0 and u[i] is not None:
                    v[ind] = costMatrix[i][ind] - u[i]
    return u,v

def findDeltas(cstMat, allMat, u,v ):
    deltas = [[None for _ in range(len(allMat[0]))] for _ in range(len(allMat))]
    for i in range(len(allMat)):
        for j in range(len(allMat[0])):
            if allMat[i][j]==0:
                deltas[i][j] = cstMat[i][j] - u[i] - v[j]

    return deltas

def isOptimal(deltas):
    for i in range(len(deltas)):
        for j in range(len(deltas[0])):
            if deltas[i][j] is not None and deltas[i][j] < 0:
                return False
    return True

def newAlloc(allMat, deltas):
    #find the most negative
    ij= [-1,-1]
    mostNeg = 1
    for i in range(len(deltas)):
        for j in range(len(deltas[0])):
            if deltas[i][j] is not None and deltas[i][j] < 0 and deltas[i][j] < mostNeg:
                mostNeg = deltas[i][j]
                ij[0] = i
                ij[1] = j

    #find loop
    allMat[ij[0]][ij[1]] = sys.maxsize
    _,elimRows,elimCols = isIndependentAllocation( allMat )
    rowinds = [i for i in range(len(elimRows)) if elimRows[i]==0]
    colinds = [i for i in range(len(elimCols)) if elimCols[i]==0]
    path = [[ij[0],ij[1]]]
    indices = [[x,y] for x in rowinds for y in colinds if allMat[x][y]!=0]
    indices.remove(path[0])
    dist = sys.maxsize
```

```
    inds = []
    n = len(indices)+1
    while len(path)!=n:
        t = len(indices)
        dist = sys.maxsize
        for i in range(t):
            d = abs(path[-1][0]-indices[i][0])+abs(path[-1][1] - indices[i][1])
            if d < dist:
                dist = d

inds.append([indices[i][0],indices[i][1]])
        path.append(inds[0])
        inds.clear()
        indices.remove(path[-1])

    #modify allocation
    val = min([allMat[path[t][0]][path[t][1]] for t in range(1,len(path),2) if allMat[path[t][0]][path[t][1]]!=0.000001])
    allMat[path[0][0]][path[0][1]] = 0
    for i in range(len(path)):
        if i%2==0:
            allMat[path[i][0]][path[i][1]] += val
        else:
            allMat[path[i][0]][path[i][1]] -= val

    #num Allocs
    numAlloc=0
    for i in range(len(allMat)):
        for j in range(len(allMat[0])):
            if allMat[i][j]>0:
                numAlloc+=1

    return allMat, numAlloc

def removeDeg(allMat, cstMat):
    for i in range(len(allMat)):
        for j in range(len(allMat[0])):
            if allMat[i][j]==0:
                allMat[i][j] = 0.000001
                isIndep = isIndependentAllocation( allMat )[0]
                if isIndep:
                    return allMat
                else:
                    allMat[i][j] = 0
    return allMat

def main():
    #1. get the cost matrix
    costMatrix = getCostMatrix()
    printMatrix('cost', costMatrix)

    #2. check if the problem is balanced
    isBal = isBalanced(costMatrix)
    if isBal:
        print('It is a balanced problem')
    else:
        print('It is an unbalanced problem')
        costMatrix = balanceProblem(costMatrix)

    #3. calculate the cost
    cost, numAllocated, allocMatrix = getTotalCost(costMatrix)
    printMatrix('allocation', allocMatrix)
    print('Calculated total cost = ',cost)

    cstMat = [x[:-1] for x in costMatrix]
    cstMat.pop()
    allMat = [x[:-1] for x in allocMatrix]
    allMat.pop()
    while 1:
        #4. check for degeneracy
        isDeg = isDegenerate(costMatrix, numAllocated)
        if isDeg:
            print('It is a degenerate solution\nMaking it a non-degenerate solution...\n')
            allMat = removeDeg(allMat, cstMat)
            numAllocated+=1
            print('\nModified Non-degenerate allocation\n')
            for i in range(len(allMat[0])):
                print('\t\tD%s'%(i+1), end='')
            print()
            for i in range(len(allMat)):
                print('S%s'%(i+1), end='')
                for j in range(len(allMat[0])):
                    print('\t\t',allMat[i][j],end='')
                print()
            print()
```

```
        continue
        return
    else:
        print('It is a non-degenerate solution')

    #5 check for independent allocation position
    isIndep = isIndependentAllocation(allMat)[0]
    if isIndep:
        print( 'The allocation positions are independent' )
    else:
        print( 'The allocation positions are not independent' )
        return


#6 calculate u and v values
    u,v = findUV( cstMat, allMat )
    print('u values = ',u)
    print('v values = ',v)

    #find delta[i,j] at unallocated positions
    deltas = findDeltas(cstMat, allMat,u,v)

    if (isOptimal(deltas)):
        print('Optimal allocation : \n')
        for i in range(len(allMat[0])):
            print('\t\tD%s'%(i+1), end='')
        print()
        for i in range(len(allMat)):
            print('S%s'%(i+1), end='')
            for j in range(len(allMat[0])):
                print('\t\t',allMat[i][j],end='')
            print()
        print()

        cost = 0
        for i in range(len(cstMat)):
            for j in range(len(cstMat[0])):
                cost = cost + cstMat[i][j]*allMat[i][j]
        print('Optimal cost = ',cost)
        return
    else:
        print('It is a non-optimal solution')
        allocMatrix, numAllocated = newAlloc(allMat, deltas)
        print('\nModified Allocation\n')
        for i in range(len(allMat[0])):
            print('\t\tD%s'%(i+1), end='')
        print()
        for i in range(len(allMat)):
            print('S%s'%(i+1), end='')
            for j in range(len(allMat[0])):
                print('\t\t',allMat[i][j],end='')
            print()
        print()


main()
```

The output obtained via the developed code is presented in Fig. 2.

Industrial Engineering Journal
ISSN: 0970-2555
Volume : 52, Issue 11, No. 3, November : 2023

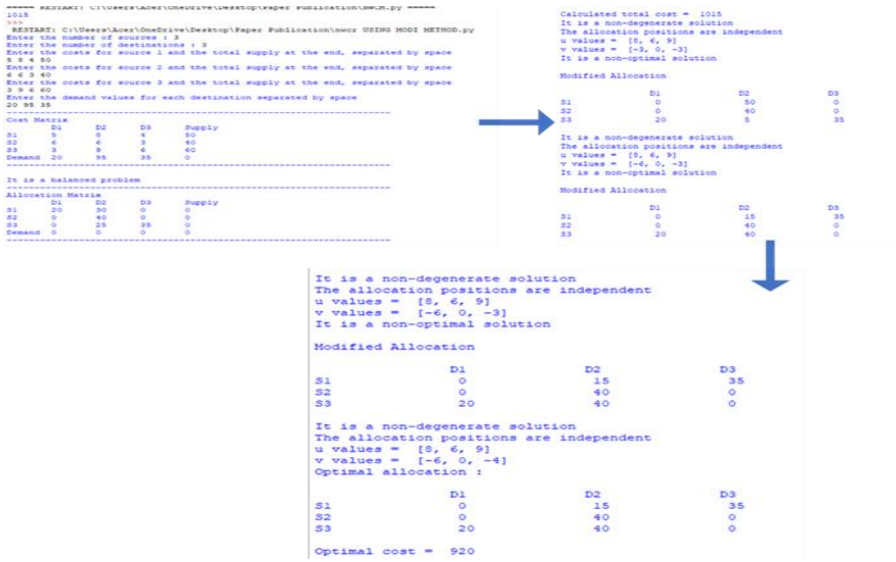

Fig. 2. Output obtained via the written code.

4. Conclusions

The following conclusions are drawn from the current study.

- The NCW rule is employed to obtain initial feasible solution to the considered transporation problem.
- Subsequently, MODI method was used optimize the initial feasible solution.
- Python language was used model the NCW and MODI methods in sequence to obtain the optimized solution for the transportation problem. The code presented reasonably good output.
- The python based method presented in this study is beneficial for logistical industry.
- New algorithm has been developed for logistical industry

References

[1]Murthy, P. Rama. Operations research (linear programming). bohem press, 2005.
[2]Klinz, Bettina, and Gerhard J. Woeginger. "The Northwest corner rule revisited." Discrete applied mathematics 159, no. 12 (2011): 1284-1289.
[3]Mishra, Shraddha. "Solving transportation problem by various methods and their comparison." International Journal of Mathematics Trends and Technology 44, no. 4 (2017): 270-275.
[4]Kanti Swarup ,P. K. Gupta ,Man Mohan . “Operation Research”, Sultan Chand & Sons, New Delhi, 2005.
[5]George, Gisha, P. Uma Maheswari, and K. Ganesan. "A modified method to solve fuzzy transportation problem involving trapezoidal fuzzy numbers." In AIP Conference Proceedings, vol. 2277, no. 1, p. 090005. AIP Publishing LLC, 2020.