Ministry of Education and Science of the Russian Federation
Peter the Great St. Petersburg State Polytechnic University
Institute of Computer Sciences and Technologies
Graduate School of Cyber-Physical Systems and Control

Report 1

Transportation Problem
Discipline: Modern Problems of Informatics and Computer Science
10 March 2017

Student Group: 13541/8

______________________ Christopher W. Blake

Professor

______________________ Rodionova E.A.

St. Petersburg
2017

Contents

Introduction

A transportation problem can be described as a supply and demand problem. There is a specific supply of product available at specified location and a certain demand required at other locations. Delivery from each possible supplier to each demander also has a different cost associated with it. Hence, the goal is to minimize the cost while still transporting all required products.

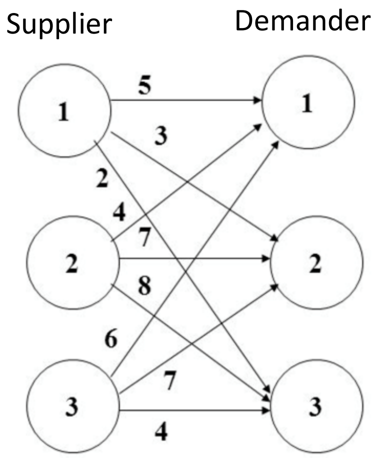


Figure 1: Transport problem in node form

Such a transportation problem is typically modeled in a node-like fashion (Figure 1). The suppliers are shown on the left and the demanders on the right. The cost associated with delivery is the connecting line. For calculation purposes, it is easier to represent this in a table (Figure 2). The suppliers and their values are shown on the left/right as “S-“. The demanders are shown along the top/bottom as “D-“. The costs associated with each delivery path are the cell intersecting an “S-“ row and “D-” column.

	D0	D1	D2	D3	D4	
S0	24	8	10	23	18	19
S1	19	1	11	9	6	14
S2	5	7	4	8	9	13
S3	6	13	3	15	5	18
	14	23	7	10	10	

Figure 2: Transport problem in table form

For solving an feasible solution to a transportation problem, three methods will be discussed, as well as the limitation of minimum delivery.

The three methods are referred to as:

1. “North West Corner” approximation method
2. “Minimum Cost Element” approximation method
3. “Least Potentials” optimization method

A sample problem is provided (# 81) from appendix 1. This sample problem is solved manually to show the process of each of the three methods. Finally, a program is demonstrated for automatically solving by these same methods, and includes the limitation of “Minimum Delivery”.

Method Verification

Balance Check

Before the transportation problem can be solved, the supply and demand must be checked such that they are balanced. This simply means that the available supply equals the amount demanded. If the problem is not balanced, additional fake rows must be added to account for the extra supply or demand.

$$isClosed = \begin{cases} true, & \sum S_i = \sum D_i \\ false, & \sum S_i <> \sum D_i \end{cases}$$

Example:

	D0	D1	D2	D3	D4	
S0	24	8	10	23	18	19
S1	19	1	11	9	6	14
S2	5	7	4	8	9	13
S3	6	13	3	15	5	18
	14	23	7	10	10	

Given the above problem table, we find the supply and demand to be equal. Hence the transportation problem is closed, and a solution can be computed.

$Supply = \sum S_i = 19 + 14 + 13 + 18 = 64$
$Demand = \sum D_i = 19 + 14 + 13 + 18 = 64$
$Supply = Demand \therefore Balanced$

Cost Computation

The total cost of delivery is calculated by multiplying the cost for each path by the number of products delivery on that path.

Example:

	D0	D1	D2	D3	D4
S0	24	8	10	23	18
	14	5	0	0	0
S1	19	1	11	9	6
	0	14	0	0	0
S2	5	7	4	8	9
	0	4	7	2	0
S3	6	13	3	15	5
	0	0	0	8	10

$Total\ Cost = 24*14 + 8*5 + 1*14 + 7*4 + 4*7 + 8*2 + 15*8 + 5*10$
$Total\ Cost = 336 + 40 + 14 + 28 + 28 + 16 + 120 + 50$
$Total\ Cost = 632$

Answer Check

The standard method to check the solution, is to compare the number of deliveries to the number of suppliers and delivers. The solution should meet the following criteria equation.

$$\#Deliveries = \#Suppliers + \#Demanders - 1$$

Example, using the previous table:
$\#Deliveries = 8,\ \#Suppliers = 4,\ \#Demanders = 5$
$8 = 4 + 5 - 1 \ \therefore True$

Method Descriptions

North West Corner Approximation

This approximation is performed by continuously finding the most northwest cell of the table and picking the min between the supplier and demander for delivery. This process is repeated for the entire grid until all supply and demand is met.

Step 1 – Find most north west cell

	D0	D1	D2	D3	D4	
S0	24	8	10	23	18	19
S1	19	1	11	9	6	14
S2	5	7	4	8	9	13
S3	6	13	3	15	5	18
	14	23	7	10	10	

1. Locate the cell that is most north west.
2. Ensure there is available supply.
3. Ensure that is available demand.

Step 2 – Choose delivery amount

	D0	D1	D2	D3	D4		
S0	24	8	10	23	18	~~19~~	5
	14						
S1	19	1	11	9	6	14	
S2	5	7	4	8	9	13	
S3	6	13	3	15	5	18	
	~~14~~	23	7	10	10		
	0						

1. Select the minimum value between the Supply and the demand. Min {14, 19} = 14
2. Set this minimum (14) to the cell.
3. Subtract the difference from the supply and demand.

Step 3 – Remove rows and columns

	D0	D1	D2	D3	D4		
S0	24	8	10	23	18	~~19~~	5
	14						
S1	19	1	11	9	6	14	
	0						
S2	5	7	4	8	9	13	
	0						
S3	6	13	3	15	5	18	
	0						
	~~14~~	23	7	10	10		
	0						

1. Mark columns columns that no longer have demand.
2. Mark rows that no longer have supply.

Step 4 - Repeat

	D0	D1	D2	D3	D4			
S0	24	8	10	23	18	~~19~~	~~5~~	0
	14	5	0	0	0			
S1	19	1	11	9	6	14		
	0							
S2	5	7	4	8	9	13		
	0							
S3	6	13	3	15	5	18		
	0							
	~~14~~	~~23~~	7	10	10			
	0	18						

1. Repeat steps 1 - 3

Step 5 - Finish

	D0	D1	D2	D3	D4				
S0	24	8	10	23	18	~~19~~	~~5~~	0	
	14	5	0	0	0				
S1	19	1	11	9	6	~~14~~	0		
	0	14	0	0	0				
S2	5	7	4	8	9	~~13~~	~~9~~	~~2~~	0
	0	4	7	2	0				
S3	6	13	3	15	5	~~18~~	~~10~~	0	
	0	0	0	8	10				
	~~14~~	~~23~~	~~7~~	~~10~~	~~10~~				
	0	~~18~~	0	~~8~~	0				
		4		0					
		0							

1. Repeat step 4 until the table is full and all supply is transported to all demanders.

Minimum Cost Element Approximation

This approximation is performed by continuously finding the available cell with minimum cost and picking the min between the supplier and demander for delivery. This process is repeated for the entire grid until all supply and demand is met.

Step 1 – Find minimum cost cell

	D0	D1	D2	D3	D4	
S0	24	8	10	23	18	19
S1	19	1	11	9	6	14
S2	5	7	4	8	9	13
S3	6	13	3	15	5	18
	14	23	7	10	10	

1. Locate the cell that has the lowest cost value.
2. Ensure there is available supply.
3. Ensure that is available demand.

Step 2 – Choose delivery amount

	D0	D1	D2	D3	D4		
S0	24	8	10	23	18	19	
S1	19	1	11	9	6	~~14~~	0
		14					
S2	5	7	4	8	9	13	
S3	6	13	3	15	5	18	
	14	~~23~~	7	10	10		
		9					

1. Select the minimum value between the Supply and the demand. Min {14, 19} = 14
2. Set this minimum (14) to the cell.
3. Subtract the difference from the supply and demand.

Step 3 – Remove rows and columns

	D0	D1	D2	D3	D4		
S0	24	8	10	23	18	19	
S1	19	1	11	9	6	~~14~~	~~0~~
		14	0	0	0		
S2	5	7	4	8	9	13	
S3	6	13	3	15	5	18	
	14	~~23~~	7	10	10		
		9					

1. Mark columns columns that no longer have demand.
2. Mark rows that no longer have supply.

Step 4 - Repeat

	D0	D1	D2	D3	D4		
S0	24	8	10	23	18	19	
			0				
S1	19	1	11	9	6	~~14~~	~~0~~
		14	0	0	0		
S2	5	7	4	8	9	13	
			0				
S3	6	13	3	15	5	~~18~~	11
			7				
	14	~~23~~	~~7~~	10	10		
		9	0				

1. Repeat steps 1 - 3

Step 5 - Finish

	D0	D1	D2	D3	D4				
S0	24	8	10	23	18	~~19~~	~~10~~	0	
	0	9	0	10	0				
S1	19	1	11	9	6	~~14~~	0		
	0	14	0	0	0				
S2	5	7	4	8	9	~~13~~	0		
	13	0	0	0	0				
S3	6	13	3	15	5	~~18~~	~~11~~	~~1~~	0
	1	0	7	0	10				
	~~14~~	~~23~~	~~7~~	~~10~~	~~10~~				
	~~1~~	~~9~~	0		0				
	0	0							

1. Repeat step 4 until the table is full and all supply is transported to all demanders.

Least Potentials Optimization

This method is an optimization method of an existing feasible solution. It calculates a penalty score for each cell. By reducing the penalty for each cell, the solution becomes optimized. This is reduced by identifying an adjustment loop, modifying by a specific amount, and repeating the process. When all penalties are zero or negative, the optimal solution has been found.

Step 1 – Create initial approximation

	D0	D1	D2	D3	D4
S0	24	8	10	23	18
	14	5	0	0	0
S1	19	1	11	9	6
	0	14	0	0	0
S2	5	7	4	8	9
	0	4	7	2	0
S3	6	13	3	15	5
	0	0	0	8	10

1. Generate an initial feasible solution using an approximation method.

Step 2 – Calculate U & V values

			V0	V1	V2	V3	V4
			24	8	5	9	-1
			D0	D1	D2	D3	D4
U0	0	S0	24	8	10	23	18
			14	5	0	0	0
U1	-7	S1	19	1	11	9	6
			0	14	0	0	0
U2	-1	S2	5	7	4	8	9
			0	4	7	2	0
U3	6	S3	6	13	3	15	5
			0	0	0	8	10

1. Identify allocated cells.
2. Assume U1 = 0
3. Solve remaining U & V values with these cells and this equation:

$$U_s + V_d = C_{sd}$$

Where:
s = supplier index
d = demander index
C = Cost

Step 3 – Calculate penalty values

			V0	V1	V2	V3	V4
			24	8	5	9	-1
			D0	D1	D2	D3	D4
U0	0	S0	24	8	10	23	18
					-5	-14	-19
U1	-7	S1	19	1	11	9	6
			-2		-13	-7	-14
U2	-1	S2	5	7	4	8	9
			18				-11
U3	6	S3	6	13	3	15	5
			24	1	8		

1. Identify unallocated cells.
2. Calculate penalty values at these cells using this equation:

$$P_{sd} = U_s + V_d - C_{sd}$$

Where:
s = supplier index
d = demander index
P = penalty
C = cost

3. Check penalty values. If all

Step 4 – Check end situation

	D0	D1	D2	D3	D4
S0	24	8	10	23	18
			-5	-14	-19
S1	19	1	11	9	6
	5		-6	0	-7
S2	5	7	4	8	9
	19				-10
S3	6	13	3	15	5
	18	-5	2		

1. Check all penalty values.
2. If all penalty values are negative or zero, then the optimal solution has been found.
3. If positive values fond, continue to step 4.

*Highlighted cells are not negative, hence not optimal solution.

Step 4 – Find start position

			V0	V1	V2	V3	V4
			24	8	5	9	-1
			D0	D1	D2	D3	D4
U0	0	S0	24	8	10	23	18
					-5	-14	-19
U1	-7	S1	19	1	11	9	6
			-2		-13	-7	-14
U2	-1	S2	5	7	4	8	9
			18				-11
U3	6	S3	6	13	3	15	5
			24	1	8		

1. Identify cell with highest penalty value.
2. Uses this cell as the start point for the next step.

Step 5* – Find calculation loop

	D0	D1	D2	D3	D4
S0	24	8	10	23	18
	14	5	0	0	0
S1	19	1	11	9	6
	0	14	0	0	0
S2	5	7	4	8	9
	0	4	7	2	0
S3	6	13	3	15	5
	0	0	0	8	10

1. Start at the highest penalty cell found in previous step.
2. *Create a loop through cells back to the start cell.
3. Turning may only occur at allocated cells.
4. Path steps must not be diagonal.

Step 6 – Identify adjustment amount

	D0	D1	D2	D3	D4
S0	-	+			
	14	5			
S1					
S2		-		+	
		4		2	
S3	+			-	
	0			8	

1. At each turn, mark the cell, alternating, with addition or subtraction. This will be the adjustment operation.
2. From the cells marked as subtraction, pick the lowest value. This will be the delivery adjustment value (dDel).

Step 7 – Adjust solution & return to step 2

	D0	D1	D2	D3	D4
S0	24	8	10	23	18
	10	9	0	0	0
S1	19	1	11	9	6
	0	14	0	0	0
S2	5	7	4	8	9
	0	0	7	6	0
S3	6	13	3	15	5
	4	0	0	4	10

1. Using the operations and adjustment value in previous step, modify the current solution. The equation should be:

$$Del_{new} = Del_{old} +/- dDel$$

Where:
Del = delivery amount
dDel = adjustment amount

***Loop Discovery Algorithm**

1. Identify all allowable movements (vertically/horizontally) from each non-empty cell to other non-empty cells.
2. Remove cells with only 1 movement possibility. These are endpoints.
3. Remove cells that do not allow turning. These are cells between other cells.
4. Repeat steps 2 and 3 until no changes occur.
5. The remaining cells are the path of the cycle.

Minimum Delivery Limitation

This limitation requires that a minimum delivery be made between a specific supplier and demander. Mathematically it has the following structure.

$$DelMin_{sd} \geq \propto$$

Where:

$DelMin_{sd}$ = Minimum deliver amount for path [s,d]
s = index of supplier
d = index of demander

This problem is solved by modifying the original problem and follows a modified solution logic, involving three steps.

1.) For each specified minimum delivery at [s,d].
   a. Remove the amount from supply
   b. Remove the amount from demand.
2.) Calculate the solution per a previously discussed method.
   (ie North West, Minimum Cost, or Least Potentials).
3.) For each specified minimum delivery amount.
   a. Add the amount to the solution at index [s,d].

Results

Manual Solution

Using the previously mentioned methods, the solution was calculated manually in excel. Below is a summary of each solution. The sum of the supply and demand

Problem Form Check:
Supply = Demand = 64 => Balanced

	D0	D1	D2	D3	D4				
S0	24	8	10	23	18	~~19~~	~~5~~	0	
	14	5	0	0	0				
S1	19	1	11	9	6	~~14~~	0		
	0	14	0	0	0				
S2	5	7	4	8	9	~~13~~	~~9~~	~~2~~	0
	0	4	7	2	0				
S3	6	13	3	15	5	~~18~~	~~10~~	0	
	0	0	0	8	10				
	~~14~~	~~23~~	~~7~~	~~10~~	~~10~~				
	0	~~18~~	0	~~8~~	0				
		4		0					
		0							

Method: North West
Cost: 632
Solution Check:
Deliveries (8) = Suppliers (4) + Demand (5) – 1 => True

	D0	D1	D2	D3	D4				
S0	24	8	10	23	18	~~19~~	~~10~~	0	
	0	9	0	10	0				
S1	19	1	11	9	6	~~14~~	0		
	0	14	0	0	0				
S2	5	7	4	8	9	~~13~~	0		
	13	0	0	0	0				
S3	6	13	3	15	5	~~18~~	~~11~~	~~1~~	0
	1	0	7	0	10				
	~~14~~	~~23~~	~~7~~	~~10~~	~~10~~				
	~~1~~	~~9~~	0		0				
	0	0							

Method: Minimum Cost
Cost: 458

	D0	D1	D2	D3	D4
S0	24	8	10	23	18
	0	12	7	0	0
S1	19	1	11	9	6
	0	11	0	3	0
S2	5	7	4	8	9
	6	0	0	7	0
S3	6	13	3	15	5
	8	0	0	0	10

Method: Least Potentials
Cost: 388
Solution Check:
Deliveries (8) = Suppliers (4) + Demand (5) – 1 => True

Software Solution

A software program with visual interface for solving transportation problems has been created. The interface can be seen on the right (Figure 3)

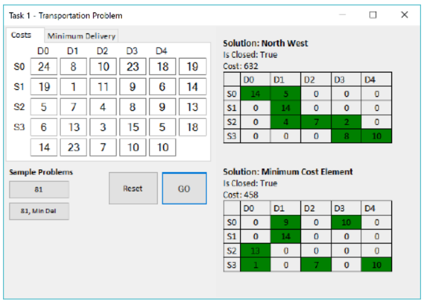


Figure 3: Transportation Problem Software

Using this program, the user may enter the cost values, supply amounts, and demand amounts into a grid. After entering the costs, the user simply presses "GO" and the solutions are displayed on the right.

<u>Minimum Delivery Requirement</u>
The user may select the second tab, where delivery requirements may be entered (Figure 4). The user simply enters values, and again presses "GO".

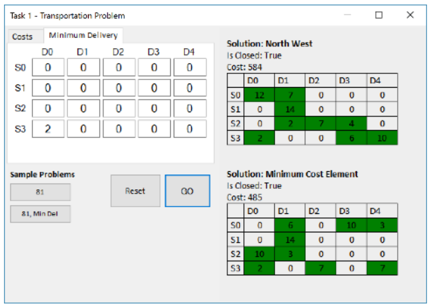


Figure 4: Minimum Delivery Adjustment

Conclusion

A transportation problem was solved by various methods, including the "North West Corner", "Minimum Cost Element" and "Least Potentials" methods. The "North West Corner" and "Minimum Cost Element" methods are used for producing an initial approximation and the "Least Potentials" method is used for further refinement. This was demonstrated manually for each method using an example problem. Finally, the above methods were programmed into a simple-to-use interface.

Comparing the solutions for example 81, the cost values show that, in this case, the "Minimum Cost Element" outperforms the "North West" method. However the cost is able to be further reduced using the "Least Potentials" optimization.

Solution Costs

North West Corner:	632
Minimum Cost Element:	458
Least Potentials:	388

Solution: North West Method
Cost: 632

	D0	D1	D2	D3	D4
S0	14	5	0	0	0
S1	0	14	0	0	0
S2	0	4	7	2	0
S3	0	0	0	8	10

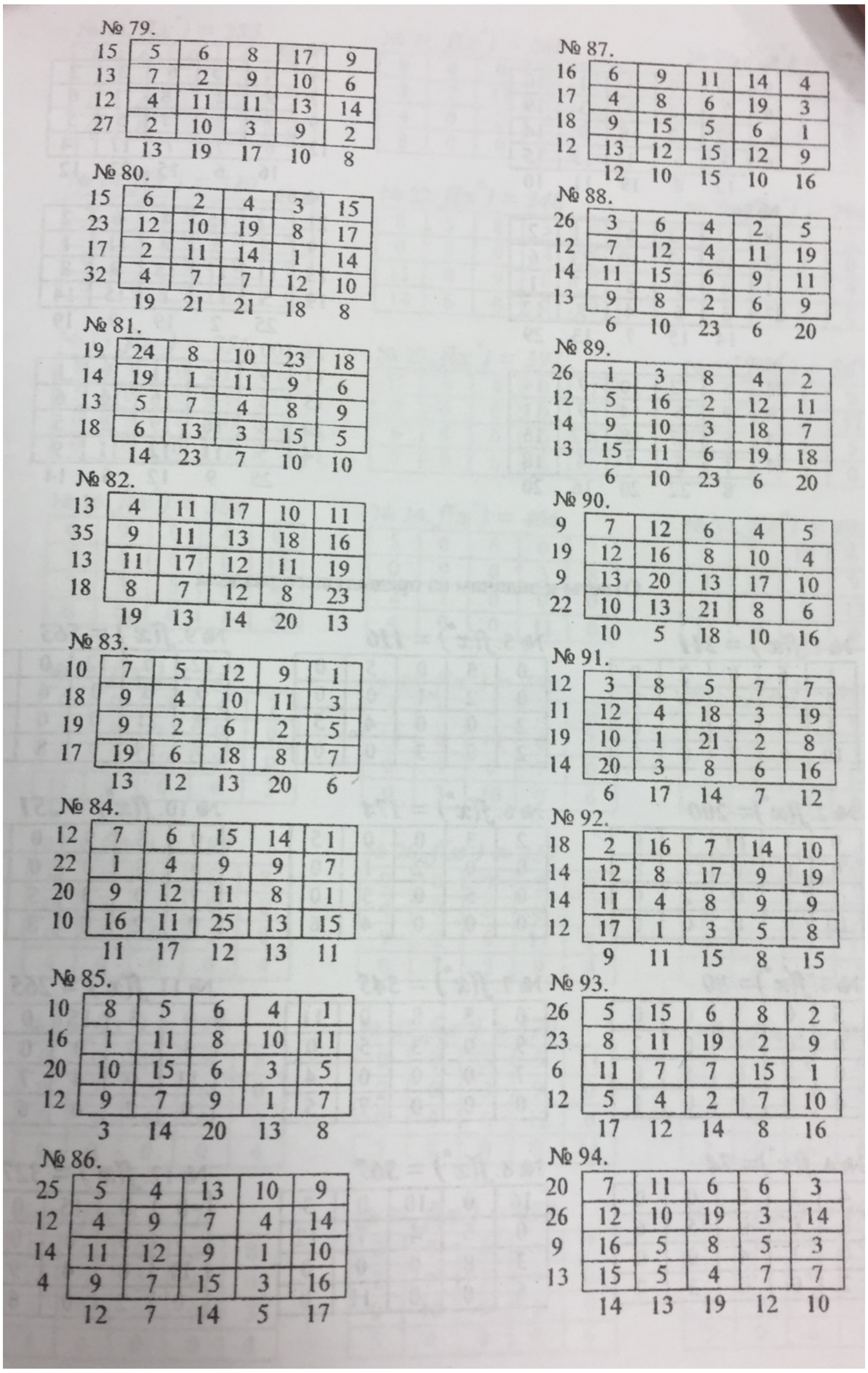
Solution: Minimum Cost Element Method
Cost: 458

	D0	D1	D2	D3	D4
S0	0	9	0	10	0
S1	0	14	0	0	0
S2	13	0	0	0	0
S3	1	0	7	0	10

Solution: Least Potentials Optimization
Cost: 388
Cycles: 4

	D0	D1	D2	D3	D4
S0	0	12	7	0	0
S1	0	11	0	3	0
S2	6	0	0	7	0
S3	8	0	0	0	10

Appendix 1 – Sample Problems



№ 79.

15	5	6	8	17	9
13	7	2	9	10	6
12	4	11	11	13	14
27	2	10	3	9	2
	13	19	17	10	8

№ 80.

15	6	2	4	3	15
23	12	10	19	8	17
17	2	11	14	1	14
32	4	7	7	12	10
	19	21	21	18	8

№ 81.

19	24	8	10	23	18
14	19	1	11	9	6
13	5	7	4	8	9
18	6	13	3	15	5
	14	23	7	10	10

№ 82.

13	4	11	17	10	11
35	9	11	13	18	16
13	11	17	12	11	19
18	8	7	12	8	23
	19	13	14	20	13

№ 83.

10	7	5	12	9	1
18	9	4	10	11	3
19	9	2	6	2	5
17	19	6	18	8	7
	13	12	13	20	6

№ 84.

12	7	6	15	14	1
22	1	4	9	9	7
20	9	12	11	8	1
10	16	11	25	13	15
	11	17	12	13	11

№ 85.

10	8	5	6	4	1
16	1	11	8	10	11
20	10	15	6	3	5
12	9	7	9	1	7
	3	14	20	13	8

№ 86.

25	5	4	13	10	9
12	4	9	7	4	14
14	11	12	9	1	10
4	9	7	15	3	16
	12	7	14	5	17

№ 87.

16	6	9	11	14	4
17	4	8	6	19	3
18	9	15	5	6	1
12	13	12	15	12	9
	12	10	15	10	16

№ 88.

26	3	6	4	2	5
12	7	12	4	11	19
14	11	15	6	9	11
13	9	8	2	6	9
	6	10	23	6	20

№ 89.

26	1	3	8	4	2
12	5	16	2	12	11
14	9	10	3	18	7
13	15	11	6	19	18
	6	10	23	6	20

№ 90.

9	7	12	6	4	5
19	12	16	8	10	4
9	13	20	13	17	10
22	10	13	21	8	6
	10	5	18	10	16

№ 91.

12	3	8	5	7	7
11	12	4	18	3	19
19	10	1	21	2	8
14	20	3	8	6	16
	6	17	14	7	12

№ 92.

18	2	16	7	14	10
14	12	8	17	9	19
14	11	4	8	9	9
12	17	1	3	5	8
	9	11	15	8	15

№ 93.

26	5	15	6	8	2
23	8	11	19	2	9
6	11	7	7	15	1
12	5	4	2	7	10
	17	12	14	8	16

№ 94.

20	7	11	6	6	3
26	12	10	19	3	14
9	16	5	8	5	3
13	15	5	4	7	7
	14	13	19	12	10

Appendix 2 – C# Code

Transport Problem Class

```
public class TransportProblem
{
    //Fields
    private double[] suppliers = null;
    private double[] demanders = null;
    private double[,] costs = null;
    private double[,] minimumDelivery = null;

    //Constructor
    public TransportProblem(int numSuppliers, int numDemanders)
    {
        this.suppliers = new double[numSuppliers];
        this.demanders = new double[numDemanders];
        this.costs = new double[numSuppliers, numDemanders];
        this.minimumDelivery = new double[numSuppliers, numDemanders];
    }

    //Properties
    public bool isReady
    {
        get
        {
            if (suppliers == null) return false;
            if (demanders == null) return false;
            if (costs == null) return false;

            //Else
            return true;
        }
    }
    public bool isBalanced
    {
        get
        {
            if (suppliers.Sum() == demanders.Sum())
                return true;
            else
                return false;
        }
    }
    public double[] Suppliers
    {
        get { return suppliers; }
        set
        {
            //check if new array matches size
            if(value != null)
            if (value.Length != suppliers.Length)
                    throw new ArgumentException("Array size must be the same.");

            //Save data
            suppliers = value;

        }
    }
    public double[] Demanders
    {
        get { return demanders; }
        set
        {
```

```
            //check if new array matches size
            if (value != null)
            if (value.Length != demanders.Length)
                throw new ArgumentException("Array size must be the same.");

            //Save Data
            demanders = value;

        }
    }
    public double[,] Costs
    {
        get { return costs; }
        set
        {
            //check if new array matches size
            if (value != null)
            if (value.GetLength(0) != costs.GetLength(0) || value.GetLength(1) !=
costs.GetLength(1))
            throw new ArgumentException("Array size must be the same.");

            //Save Data
            costs = value;
        }
    }
    public double[,] MinimumDelivery
    {
        get { return minimumDelivery; }
        set
        {
            //check if new array matches size
            if (value != null)
            if (value.GetLength(0) != costs.GetLength(0) || value.GetLength(1) !=
costs.GetLength(1))
                throw new ArgumentException("Array size must be the same.");

            //Save Data
            minimumDelivery = value;

        }
    }
}
```

North West Corner Approximation Method

```
public double[,] solveNorthWest()
{
    return solveNorthWest(true);
}
public double[,] solveNorthWest(bool enableLimations)
{
    //Switch for limations modifications
    if (enableLimations)
    {
        //Account for minimum delivery
        adjustMinimumDelivery_FromSupplyAndDemand(false); //Subtract away
    }

    //Create temporary variables
    double[] sup = (double[]) suppliers.Clone();
    double[] dem = (double[]) demanders.Clone();
    double[,] solution = new double[suppliers.Length, demanders.Length];

    //Cycle through each solution position
    int s = 0;
    int d = 0;
    while (s < sup.Length && d < dem.Length)
    {
        //Get min of supply and demand
        double min = (new double[] { sup[s], dem[d] }).Min();

        //Set to solution
        solution[s, d] = min;

        //Remove from supply and demand
        sup[s] -= min;
        dem[d] -= min;

        //Find next most northwest position
        try
        {
            while (sup[s] == 0) { s++; }
            while (dem[d] == 0) { d++; }
        }
        catch
        {
            //All finished
            break;
        }
    }

    //Switch for limations modifications
    if (enableLimations)
    {
        //Account for minimum delivery
        addMinimumDelivery_ToSolution(solution);
        adjustMinimumDelivery_FromSupplyAndDemand(true); //Add back
    }

    //Return the results
    return solution;
}
```

Minimum Cost Element Approximation Method

```
public double[,] solveMinimumCostElement()
{
    //Account for minimum delivery
    adjustMinimumDelivery_FromSupplyAndDemand(false); //Subtract away

    //Create temporary variables
    double[] sup = (double[])suppliers.Clone();
    double[] dem = (double[])demanders.Clone();
    double[,] solution = new double[suppliers.Length, demanders.Length];

    //Get dimensions
    int rows = solution.GetLength(0);
    int cols = solution.GetLength(1);

    //Cycle through each solution position
    while (sup.Sum() > 0 && dem.Sum() > 0)
    {
        //Find min cost position, that has no solution value
        double currMinCost = double.PositiveInfinity;
        int rMin = 0;
        int cMin = 0;
        for (int r = 0; r < rows; r++)
            for (int c = 0; c < cols; c++)
            {
                //Skips finished solutions
                if (solution[r,c] != 0) { continue; }
                if (sup[r] == 0) { continue; }
                if (dem[c] == 0) { continue; }

                if (costs[r,c] < currMinCost)
                {
                    rMin = r;
                    cMin = c;
                    currMinCost = costs[r, c];
                }
            }
        int s = rMin;
        int d = cMin;

        //Get min of supply and demand
        double min = (new double[] { sup[s], dem[d] }).Min();

        //Set to solution
        solution[s, d] = min;

        //Remove from supply and demand
        sup[s] -= min;
        dem[d] -= min;
    }

    //Account for minimum delivery
    addMinimumDelivery_ToSolution(solution);
    adjustMinimumDelivery_FromSupplyAndDemand(true); //Add back

    return solution;
}
```

Least Potentials Optimization Method

```
public double[,] solveUV(out int cycles)
{
    //Account for minimum delivery
    adjustMinimumDelivery_FromSupplyAndDemand(false); //Subtract away

    //Get dimensions
    int rows = Costs.GetLength(0);
    int cols = Costs.GetLength(1);

    //Get Northwest approximation
    double[,] solutionCurr = solveNorthWest(false);

    //Cycle until end condion met
    cycles = 0; //For statistics
    while (true)
    {

        #region Calculate UV values
        double[] u;
        double[] v;
        calculateUV_Values(solutionCurr, out u, out v);
        #endregion

        #region  Calculate penalty values, track location of greatest penalty
        double[,] penalties = new double[rows, cols];
        int rMax = -1;
        int cMax = -1;
        double penaltyMax = double.NegativeInfinity;
        bool allNegative = true;

        //Calculate penalties
        for (int r = 0; r < rows; r++)
        {
            for (int c = 0; c < cols; c++)
            {
                //Calculate only for unassigned cells
                if (solutionCurr[r, c] == 0)
                {
                    //Get and store value
                    double penalty = u[r] + v[c] - Costs[r, c];
                    penalties[r, c] = penalty;

                    //Check sign
                    if (penalty > 0) allNegative = false;

                    //Check for max
                    if (penalty > penaltyMax)
                    {
                        penaltyMax = penalty;
                        rMax = r;
                        cMax = c;
                    }
                }
            }
        }
        #endregion

        //Check end condion
        if (allNegative)
        {
            //Finished
            break;
        }

        #region Generate new iteration of solution
        //Identify loop
```

```
        int rLoopStart = rMax;
        int cLoopStart = cMax;
        int[,] loop = findLoop(solutionCurr, rLoopStart, cLoopStart);

        //Get lowest number in "negative" group (odd entries of loop)
        double minValue = double.PositiveInfinity;
        for (int p = 0; p < loop.GetLength(0); p++)
        {
            //Get cell value
            int r = loop[p, 0];
            int c = loop[p, 1];

            //Determine current operation
            if (p % 2 == 1) //odd (negative operation numbers)
            {
                if (solutionCurr[r, c] < minValue)
                    minValue = solutionCurr[r, c];
            }
        }

        //Adjust current solution
        for (int p = 0; p < loop.GetLength(0); p++)
        {
            //Get cell value
            int r = loop[p, 0];
            int c = loop[p, 1];

            //Determine current operation
            if (p % 2 == 0) //even or zero
            {
                //Add the minimum value to the solution
                solutionCurr[r, c] += minValue;
            }
            else //odd
            {
                //Remove the minimum value from the solution
                solutionCurr[r, c] -= minValue;
            }
        }
        #endregion

        cycles++;
    }

    //Account for minimum delivery
    addMinimumDelivery_ToSolution(solutionCurr);
    adjustMinimumDelivery_FromSupplyAndDemand(true); //Add back

    return solutionCurr;
}
```

Least Potentials Optimization – Support Methods

```
private void calculateUV_Values(double[,] solution, out double[] u, out double[] v)
{
    //Get dimensions
    int rows = solution.GetLength(0);
    int cols = solution.GetLength(1);

    //Result variables
    double?[] U = new double?[rows];
    double?[] V = new double?[cols];

    //Assume U0 = 0 for row 0, Solve for
    U[0] = 0;

    //Repeat loop until all values of U and V are solved
    while (true)
    {
        #region Check if U and V finished
        //Check U values
        bool uFinished = true;
        for (int i = 0; i < U.Length; i++)
        {
            if (U[i] == null)
            {
                uFinished = false;
                break;
            }
        }

        //Check V values
        bool vFinished = true;
        for (int i = 0; i < V.Length; i++)
        {
            if (V[i] == null)
            {
                vFinished = false;
                break;
            }
        }

        //Both finished
        if (uFinished && vFinished) break;
        #endregion

        //Try to solve for V values
        for (int r = 0; r < rows; r++)
        {
            //If U not set, V cannot be determined
            if (U[r] == null)
                continue;

            //Set values of V for assigned cells
            for (int c = 0; c < cols; c++)
            {
                //If already set, move to next
                if (V[c] != null)
                    continue;

                //Set the value
                if (solution[r, c] > 0)
                    V[c] = Costs[r, c] - U[r];
            }
        }

        //Try to solve for U values
        for (int c = 0; c < cols; c++)
        {
```

```
            //If V not set, U cannot be determined
            if (V[c] == null)
                continue;

            //Set values of U for assigned cells
            for (int r = 0; r < rows; r++)
            {
                //If already set, move to next
                if (U[r] != null)
                    continue;

                //Set the value
                if (solution[r, c] > 0)
                    U[r] = Costs[r, c] - V[c];
            }
        }
    }

    //Return results
    u = new double[rows];
    for (int i = 0; i < rows; i++)
        u[i] = (double)U[i];

    v = new double[cols];
    for (int i = 0; i < cols; i++)
        v[i] = (double)V[i];
}
private int[,] findLoop(double[,] solution, int rStart, int cStart)
{
    //Get dimensions
    int rows = solution.GetLength(0);
    int cols = solution.GetLength(1);

    #region Generate possible directions
    //Add temporary value at start point (so it is included in directions generation)
    solution[rStart, cStart] = 1;

    //Compute possible directions at each cell
    List<int[]>[,] allowedDirections = new List<int[]>[rows, cols];
    for (int r = 0; r < rows; r++)
    {
        for (int c = 0; c < cols; c++)
        {
            //Ignore unassigned cells
            if (solution[r, c] == 0)
                continue;

            //If not created yet, create it
            allowedDirections[r, c] = new List<int[]>();

            //Check row
            for (int cCurr = 0; cCurr < cols; cCurr++)
            {
                if (cCurr != c & solution[r, cCurr] > 0)
                {
                    allowedDirections[r, c].Add(new int[] { 0, cCurr - c });
                }
            }

            //Check colum
            for (int rCurr = 0; rCurr < rows; rCurr++)
            {
                if (rCurr != r & solution[rCurr, c] > 0)
                {
                    allowedDirections[r, c].Add(new int[] { rCurr - r, 0 });
                }
            }
```

```
        }
    }

    //Remove temporary value at start point
    solution[rStart, cStart] = 0;
    #endregion

    #region Remove bad directions
    bool changeFound = true;
    while (changeFound)
    {
        //Assume no change first
        changeFound = false;

        //Remove items with one entry
        for (int r = 0; r < rows; r++)
        {
            for (int c = 0; c < cols; c++)
            {
                List<int[]> ad = allowedDirections[r, c];
                if (ad != null && ad.Count == 1)
                {
                    //Get the entry
                    int[] dir = ad.First();

                    //Get the cell where it points, and remove the negative version
                    allowedDirections[r + dir[0], c + dir[1]].RemoveAll(i => i[0] == -
dir[0] && i[1] == -dir[1]);

                    //Remove this list
                    allowedDirections[r, c] = null;

                    //Allow another loop
                    changeFound = true;
                }
            }
        }

        //Remove items that can't turn
        for (int r = 0; r < rows; r++)
        {
            for (int c = 0; c < cols; c++)
            {
                List<int[]> ad = allowedDirections[r, c];
                if (ad != null)
                {
                    //Check if both row and column movement exist
                    int countColumnMovement = ad.FindAll(dir => dir[0] == 0).Count;
                    int countRowMovement = ad.FindAll(dir => dir[1] == 0).Count;

                    //If column move
                    if (countColumnMovement == 0 || countRowMovement == 0)
                    {
                        //For each entry
                        foreach (int[] dir in ad)
                        {
                            //Get the cell where it points, and remove the negative version
                            allowedDirections[r + dir[0], c + dir[1]].RemoveAll(i => i[0]
== -dir[0] && i[1] == -dir[1]);
                        }

                        //Remove this list
                        allowedDirections[r, c] = null;

                        //Allow another loop
                        changeFound = true;
                    }
```

```
                }

            }
        }
    }
    #endregion

    #region Generate Path
    //Start at specified position
    List<int[]> path = new List<int[]>();
    path.Add(new int[] { rStart, cStart });
    int rCurrr = rStart;
    int cCurrr = cStart;

    //Start loop
    while (true)
    {
        //Get current directions
        List<int[]> directionsCurr = allowedDirections[rCurrr, cCurrr];

        //Get first entry
        int[] dir = directionsCurr.First();

        //Move to this entry
        rCurrr += dir[0];
        cCurrr += dir[1];
        directionsCurr = allowedDirections[rCurrr, cCurrr];

        //Check if back at start
        if (rCurrr == rStart && cCurrr == cStart)
            break;

        //Add to path
        path.Add(new int[] { rCurrr, cCurrr });

        //Remove negative reference, so as not to get sent back
        allowedDirections[rCurrr, cCurrr].RemoveAll(i => i[0] == -dir[0] && i[1] == -
dir[1]);
    }
    #endregion

    #region Convert path
    //Copy from list to 2d array
    int[,] loop = new int[path.Count, 2];
    for (int i = 0; i < path.Count; i++)
    {
        //copy row value
        loop[i, 0] = path[i][0];

        //copy column value
        loop[i, 1] = path[i][1];
    }
    #endregion

    return loop;
}
```