Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of System Engineering

Bachelor Thesis

Step-by-step Solver for Transportation Problems

Tugay Demir

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Tugay Demir

Informatics

Thesis title

Step-by-step Solver for Transportation Problems

Objectives of thesis

The main objective of the bachelor thesis is to develop an application that would show the solution of classical transportation problem step-by-step and which could serve as an educational tool for students.

Methodology

The thesis will consist of two parts and those parts are theoretical part and practical part. The theoretical part will describe the mathematical formulation of transportation problem and methods that solve it. In addition, fundamentals of given programming language used in the practical part will be described.

The practical part will describe the software solution approach to solving the transportation problem step-by-step and it will be accompanied with examples. The limitations of software will be discussed in the end.

The proposed extent of the thesis

40

Keywords

transportation problem; step-by-step solution; application

Recommended information sources

Hillier, F. B. S. N. (2022). Introduction to Operations Research (9th ed.). McGraw Hill Education Private Limited; (29 September 2011).

Sierra, K., Gee, T., & Bates, B. (2022). Head First Java: A Brain-Friendly Guide (3rd ed.). O'Reilly Media.

Transportation Problem: A Special Case for Linear Programming Problems. (2002). Performance Excellence in the Wood Products Industry.

Expected date of thesis defence

2022/23 SS – FEM

The Bachelor Thesis Supervisor

Ing. Robert Hlavatý, Ph.D.

Supervising department

Department of Systems Engineering

Electronic approval: 30. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Head of department

Electronic approval: 30. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Dean

Prague on 15. 03. 2024

Declaration

I declare that I have worked on my bachelor thesis titled "Step-by-step Solver for Transportation Problems" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on date of submission _____

Acknowledgement

I would like to thank Ing. Robert Hlavatý, Ph.D. for his guidance and support throughout my Bachelor Thesis. His expertise and advice have been instrumental in helping me write my thesis. I would also like to thank all my teachers, colleagues and family who have helped me throughout my education life.

Step-by-step Solver for Transportation Problems

Abstract

The bachelor thesis aimed to show steps of the Transportation Problem and help students understand the transportation problem.

Operations research is a method developed to find the optimal solution for a particular purpose in a situation where certain constraints exist. The transportation problem is a part of operations research and aims to move products from sources to destinations at the least cost.

The theoretical part describes the transportation problem, its emergence and how it has developed until today. This part will also shed light on how it can be solved.

The practical part will explain the Java programming language and where it is used, and show the development steps of the implementation. Application testing is included in the section that follows the implementation.

Keywords: transportation problem; step-by-step solution; application

Řešení problémů s dopravou krok za krokem

Abstrakt

Bakalářská práce si kladla za cíl ukázat kroky dopravního problému a pomoci studentům pochopit dopravní problém.

Operační výzkum je metoda vyvinutá k nalezení optimálního řešení pro konkrétní účel v situaci, kdy existují určitá omezení. Problém dopravy je součástí operačního výzkumu a jeho cílem je přesun produktů ze zdrojů do destinací s co nejnižšími náklady.

Teoretická část popisuje dopravní problém, jeho vznik a jak se vyvíjel až do současnosti. Tato část také osvětlí, jak to lze vyřešit.

Praktická část vysvětlí programovací jazyk Java a kde se používá a ukáže vývojové kroky implementace. Testování aplikace je zahrnuto v části, která následuje po implementaci.

Klíčová slova: dopravní problém; řešení krok za krokem; aplikace

Table of content

1.1 List of Images 11

2 Objectives and Methodology 13

2.1 Objectives 13

2.2 Methodology 13

3 Transportation Models 14

3.1 The Mathematical Form of Transportation Problem 15

3.2 Solution of the Transportation Model 16

3.2.1 Nort-West Corner Method 18

3.2.2 Least Cost Method 18

3.2.3 Vogel's Approximation Method 19

3.2.4 Russel's Approximation Method 19

3.3 Optimal Solution for Transportation Problem 20

3.3.1 Stepping Stone Method 20

3.3.2 Modified Distribution Method 20

3.4 Special Cases of the Transportation Problem 21

3.4.1 Degenerate Cases 21

3.4.2 Unbalanced Cases 22

3.4.3 Empty Cells 22

3.4.4 Multiple Allocations 23

3.5 Transportation Problem in a Network 23

3.5.1 Optimal Path Problem 23

3.5.2 Maximal Flow Problem 24

4 Java 25

5 Practical Part 27

5.1 Application idea 27

5.1.1 Target Audience Analysis 27

5.1.2 Application Overview 27

5.1.3 Alternative Solutions 27

5.1.4 Requirement Analysis 28

SWOT analysis 28

Use Case Analysis 29

5.2 Implementation 31

5.2.1 IntelliJ IDEA Installation 31

5.2.2 IntelliJ IDEA Configuration 31

5.2.3 Starting a New IntelliJ IDEA Project 31

5.2.4 Implementation of Project 32
Transportation Class 32
5.3 Parts of the Coding 35
5.3.1 Example Solution with Transportation Solver App 38
6 Conclusion 45
References 46
Appendix 49

List of Images

1. Image 1. Transportation Model Solution Steps…… 18
2. Image 2. Solution with North-West Method……..…19
3. Image 3: Step-by-Step Transportation Solver Use Case Model……30
4. Image 4: Entrance page of step-by-step solver …….38
5. Image 5: Creation of 2x2 Transportation table in the application ……38
6. Image 6: Filling the transportation table in the application ……..38
7. Image 7:Problem solved message in the Step-by-Step solver of application……39
8. Image 8: Northwest Corner Method solution in the application……...40
9. Image 9: Solution of Least Cost Method in the application…….41
10. Image 10: Solution of Vogel's Approximation Method in the application……...42
11. Image 11: Solution of Russels's Approximation method in the application……43

1. Introduction

In its simplest form, the goal of transportation models can be defined as achieving the most cost - effective distribution of goods from various sources to different destinations. The transportation model, a specific case of the linear programming model, can be solved using the simplex approach, but using the model's unique techniques to get there is far more time and effort efficient. Frank L. Hitchcock presented the first study, which is close to the current framework of the model, in 1941. In this work, a new mathematical model was created, and a solution was suggested.

The transportation model can be applied not just to distribution planning problems, but also to problems that can be handled by the same techniques but are unrelated to transportation as a subject. The model also has several expansions, and with these extensions, the assignment of jobs to machines, people to jobs, or the path taken by a traveling salesman may be simply determined using special algorithms. The goal of the model, which generates such a diverse range of solutions, is to discover the lowest possible transportation cost and the distribution plan that achieves this lowest cost.

Companies these days must pursue growth in order to adapt ever to competition conditions, and those who fail to do so are decreasing in size. It is critical for growing organizations to minimize their expenses, and the relevance of transportation costs, particularly for manufacturing companies, reaches critical levels for the company's survival. As a result of the usage of transportation model techniques in related industries, companies' transportation expenses are greatly lowered.

1 Objectives and Methodology

1.1 Objectives

This thesis aims to develop a user-friendly, straightforward computer tool that assists individuals in solving transportation problems. Businesses and educational institutions frequently work on transportation problems to determine the most cost-effective way to move items from one location to another.

This thesis will first examine the nature of transportation problem in detail. It will discuss the history of these issues and their continued significance. This section will use books and articles to demonstrate how these issues have been resolved both historically and currently.

Following that, using the computer language Java, the thesis's primary objective is to create a program. This program will guide you through solving a transportation issue step-by-step. It will employ many approaches to problem-solving and demonstrate which one works best.

The work will also assess the program's usability and effectiveness. It seeks to ensure that students studying these topics can utilize the curriculum to improve their understanding and work through the issues independently.

By doing all of this, the thesis hopes to help students and others learn more about transportation problems and make it easier for them to find solutions.

1.2 Methodology

In operational research, the transportation problem is an important concept that centers on the effective distribution of resources to move things between several origins and various destinations. It involves decreasing transportation expenses while meeting requirements related to supply and demand.

While individuals may encounter challenges in comprehending this subject matter, often finding the learning process time-consuming, the primary objective of this thesis is to streamline and facilitate a more accessible understanding of transportation problems.

To achieve this, the study will consist of two parts for better understanding. In the first part, a literature review will be conducted on transportation problems, enabling individuals to understand both the historical and theoretical aspects of transportation problems. This will be achieved by consulting necessary sources and experiments, providing examples from them to facilitate a clearer understanding of the topic. At the end of the first part, the second part will explain the Java programming language to be used in the practical section and why it is chosen.

In the second part, the necessity of the application and the areas in which it will be used will be explained, along with the stages involved in creating the application. Following this, analyses will be conducted in the second part to emphasize the aspects that can be improved and the adequate sections. The resulting application will assist users in understanding transportation problems easily and solving them, while also explaining the solution steps in detail."

2 Transportation Models

Leonid Vitalyevich Kantorovich completed first study on transportation models in history in 1939. In this study, which examines the assignment of work to machines, costs vary based on the distribution of jobs and machines. Kantorovich has presented an incomplete algorithm for similar issues, despite its utility. Maybe this explains why his work at the time managed to garner little attention. In 1942 and 1948, he conducted research into the continuous states of transportation models. (Dantzig, 2016)

In his article titled "The Distribution of a Product from Several Sources to Numerous Localities," Frank L. Hitchcock for the first time constructed the mathematical model and proposed a solution in 1941. Tjalling C. Koopmans later dealt with the subject in his 1949 work "Optimum Utilization of the Transportation System," independently of Hitchcock's study and in greater detail. Following the release of George B. Dantzig's article titled "Application of the Simplex Method to a Transportation Problem," the most significant improvement in the transportation model was achieved. A. Charnes and W.W. Cooper devised the Stepping Stone Method in 1953, which provides a more methodical approach to Dantzig's solution. R.O. Ferguson created the Modi Method in 1955 as an alternative to the Stepping Stone Method.

Transportation models are a part of linear programming models. The model's goal is to move commodities from origin to destination at the lowest possible cost. In order to achieve this goal, the supply in the sources must be thoroughly distributed, and the demand in the targets must be totally met.

The simplex method is commonly used to solve linear programming models. Because the transportation model is a part of linear programming, it can be solved using this method. Nevertheless, if a transportation problem with m supply and n demand centers is to be handled using the simplex approach, artificial variables must be added to the model, resulting in numerous variables and operations with a huge number of rows and columns in the simplex table. This will result in a lot of effort as well as a wasted amount of time.

Due to their unique characteristics, some linear programming problems can be handled using techniques that are more effective than the simplex method. The transportation model's special structure has led to the development of distinct solution methods. The problem can be solved more quickly thanks to these different solutions.

The simplex method can handle every transportation issue, despite taking a lot of time and effort. However, since the assumptions of the transportation model are more restrictive than the linear programming assumptions, not every linear programming problem solved by the simplex method can be solved by the transportation algorithm (Özkan, 2005).

2.1 The Mathematical Form of Transportation Problem

The components of a transportation model are m sources that provide a_i (i=1,2,...,m) unit of homogenous commodities and n destinations that need b_j (j=1,2,....,n) unit goods. In this case, a_i and b_j are represented by positive integer numbers. C_{ij} stands for the amount of products delivered from the *i*th source to the *j*th destination, and this value must be known for both *i* and *j*.

Creating an integer distribution plan with the lowest total transportation cost is the goal of the model.

When defining the model, it is assumed that total supply equals total demand;

$$\sum_{i=1}^{m} ai \equiv \sum_{j=1}^{n} bj \qquad \ldots(1)$$

Where:

- a_i represents the amount provided by the *i*th source
- b_j represents the amount required by the *j*th destination

If total balance and total demand is equal as shown in equation the model is called balanced model. If total balance and total demand is not equal model called unbalanced model. When a problem came up with an unbalanced model, the model must first be converted to a balanced model before solving it (Operations Research: Principles and Practice, 1976).

The simplex method, hence the simplex table, is used in solving linear programming problems. In the solution of transportation models, the transportation table should be prepared first. The intended use of the transportation table is the same as that of the simplex table; It is to ensure that all information and data related to the problem are seen together in the same frame (Kirkpatrick, 1965). In other words, the purpose of the table is to facilitate the solution steps by summarizing all the necessary data related to the problem in a suitable way (Render & Stair, 2016).

2.2 Solution of the Transportation Model

The simplex method and the transportation problem's solution logic are similar. The iteration phases of the transportation model are essentially a simplified version of the iteration stages of the simplex technique, despite the fact that this parallelism is not very obvious due to the difference in the table that was constructed. This ease of use is a benefit of the unique structure of the transportation concept. In transportation problems, the corner points corresponding to the fundamentally appropriate solutions are explored, and the corner points are navigated until the ideal solution is found, just like in the simplex technique.

For the solution of any transportation problem, first the transportation table is prepared, and the data is placed in the table. The supply and demand totals in the table are checked. If there is no equality between aggregate supply and aggregate demand, equality should be achieved at this stage. That is, if an unbalanced model is encountered, the model should be converted to a balanced model (Ignizio et al., 1983).

In the balanced model, taking into account the row and column equivalences, the initial basic feasible solution is obtained by distributing $m + n - 1$ cells (Kara, 1991).

After finding the initial basic suitable solution, an optimality test should be used to determine whether the answer is the best. If the ideal answer is not found, iterations should be repeated again while applying other procedures until the ideal result is found. The delivery table's empty cells are looked at in order to determine whether or not the initial basic suitable solution achieved here is ideal. In order to determine whether or not the solution may be improved, each empty cell is examined to see if a better solution than the initial basic suitable answer can be found in the event that any empty cell is loaded.

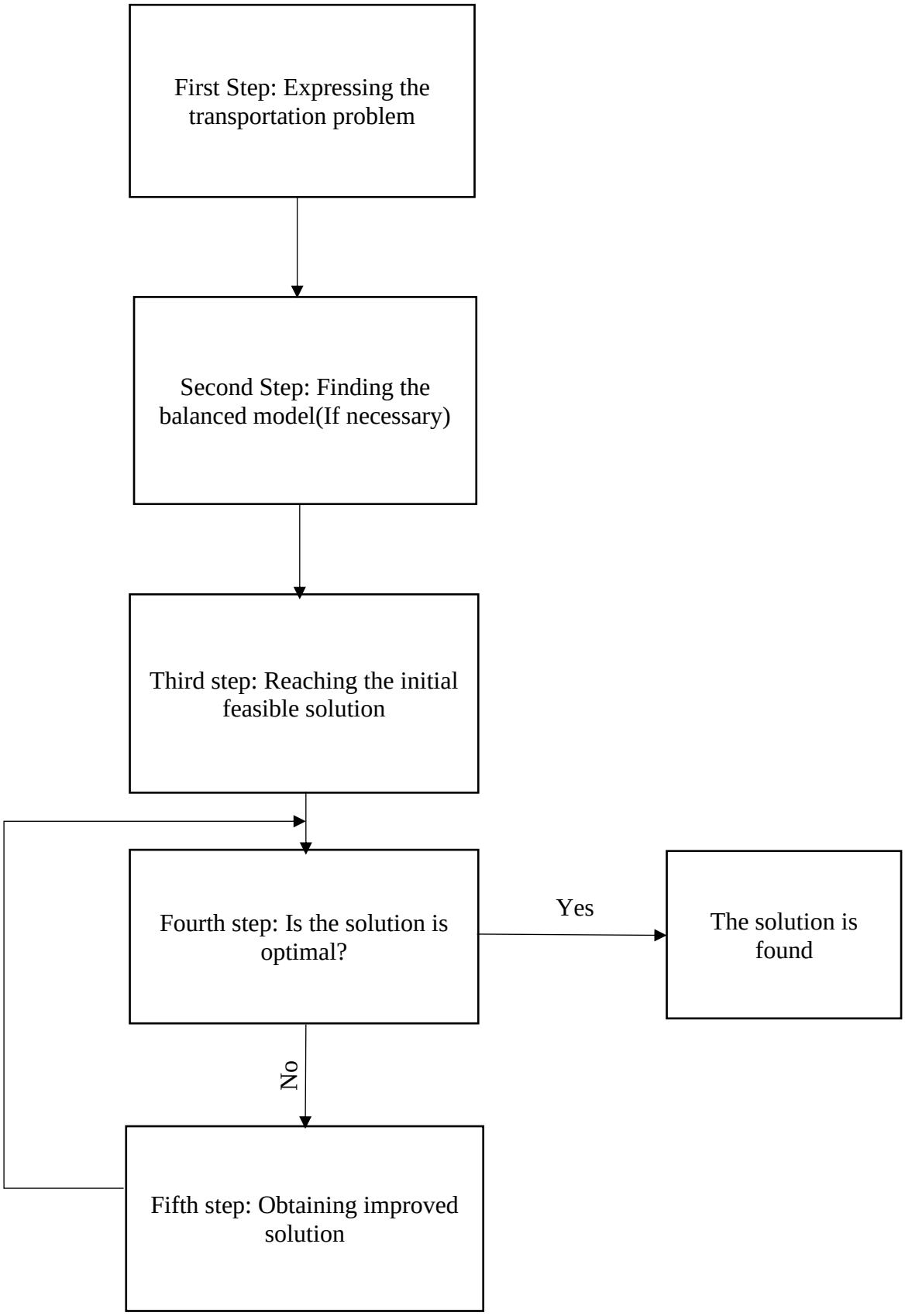


Image 1: Transportation Model Solution Steps (Source: Own Material)

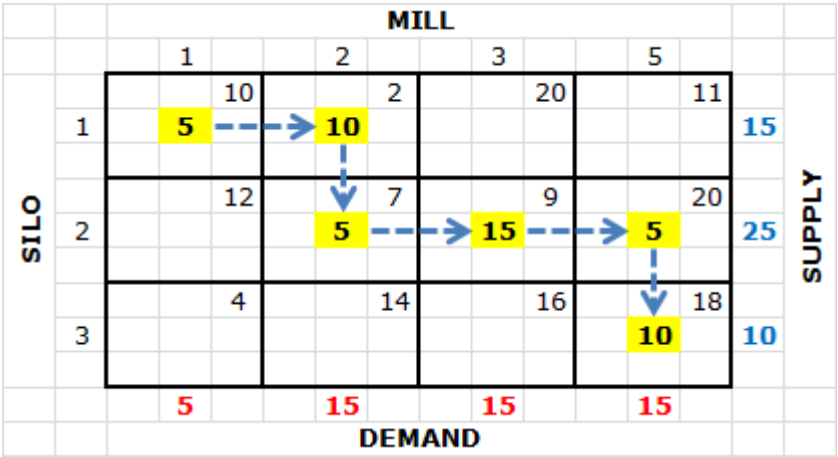
2.2.1 Nort-West Corner Method

This approach is one of several that methodically looks for answers to transportation model problems. The solution of the problem is called the North-West Corner Method because it starts from the cell in the northwest corner of the prepared delivery table(Tulunay,1987). George B. Dantzig introduced the method, while Charnes and Cooper gave it a name.

The steps of this method can basically be listed as follows:

1. Assign as many as possible to the chosen box, then subtract this number from the supply and demand quantities before making the required changes.
2. Cancel a row or column that has zero supply or demand as a result of subtraction to prevent future reassignment. If both the row and column are zero at the same time, choose one and ignore the zero supply (demand) in the non-cancelled row (column).
3. Stop at the row or column that has not been canceled. Otherwise, if the column was canceled in the previous operation, move to the right box, if the row is canceled, move to the next box. Return the first step.

Image 2 : Solution with North-West Method



(Source: https://www.linearprogramming.info/northwest-corner-method-transportation-algorithm-in-linear-programming/)

2.2.2 Least Cost Method

Despite the fact that the lowest cost cell method is not optimum because of its use of costs, it is possible to achieve a lower cost solution compared to the north-west corner method. Therefore, the results to be obtained with this method will be encountered with fewer iterations at the optimality test stage compared to the north-west corner method(Levin et al., 1993).

The steps of this method can basically be listed as follows:

1. Pick the cell with the lowest unit cost c_{ij} and assign as much as possible to it, i.e.
2. Reduce this minimum value from the supply and demand. If the supply s_i is 0 and the demand d_j is 0, cross off that row and column, respectively. If the minimum unit cost cell is not unique, select the cell with the highest allocation potential.
3. Continue these steps for each uncrossed row and column until all supply and demand values are equal to zero.

2.2.3 Vogel's Approximation Method

The VAM Method, which was proposed by William R. Vogel in 1958, gives a much better initial basis suitable solution than the north-west and lowest cost cell methods, and sometimes even the optimal solution (especially the direct optimal solution for small-sized problems) (Kara, 1991).

In general, the VAM Method is based on the idea of determining the missed opportunities per unit and avoiding missing the greatest opportunity, if loading is not performed in the lowest cost cell but in the second lowest cost cell throughout the table.

The steps of this method can basically be listed as follows:

1. Determine the two lowest expenses in each column and row. Determine the row and column differences.
2. Select the row or column with the greatest cost difference and assign the maximum possible number of units to the least cost route in that row or column.
3. If the assignment in step two meets the requirement at that destination, the related column is deleted. Otherwise, delete the corresponding row when the supply at the origin is cleared.
4. If every supply is spent and every demand met stop. Or, return to step 1.

2.2.4 Russel's Approximation Method

With the RAM Method - just like with the VAM Method - it is possible to arrive at a good initial solution, sometimes even a very close to optimal solution or directly to the optimal solution.

This method includes more calculation processes than the VAM Method. Although there is no definite opinion as to which of the two methods gives the better solution, it is claimed in some sources that a better result is obtained with the RAM Method (G. B. Dantzig, n.d.).

1. Determine the U_i for each source row remaining under evaluation (largest cost in row i).
2. Determine the V_j value for each destination column still under consideration (largest cost in column j).
3. For each variable, calculate $\Delta ij = cij - (\bar{}Ui + \bar{}Vj)$.
4. Choose the variable with the highest negative value and arbitrarily break ties.
5. Assign the maximum amount. Omit necessary cells from consideration. Turn back to Step-1.

2.3 Optimal Solution for Transportation Problem

In transportation models, after obtaining the initial basic feasible solution, this solution is optimal; The lowest cost result should be tested under the same restrictive conditions. When performing the test, all empty cells—non-essential variables—should be examined first. If the total cost is lower when any empty cell is used, it is determined that the initial basic feasible solution is not the ideal solution, and the non-basic variable that gives the cost decrease should be used as the basis If none of the empty cells have a beneficial effect on the cost, the solution achieved with the original basic appropriate solution is optimal, it is the lowest cost.

Two methods are used when testing optimality:

- Stepping Stone Method
- Modi Method

There is a point to be considered before applying the optimality test with both methods. m=number of sources, n=number of targets, the number of cells involved in the initial basic solution should be m+n-1 (Meredith, 1994).

If this equality is not met, it will be impossible to make some calculations while applying both methods. Therefore, in such a case, the imbalance should be eliminated first, and the optimality test should be applied after this correction process.

2.3.1 Stepping Stone Method

This method, which tests the optimality and provides the best solution step by step, was developed by A. Charnes and W.W.Cooper (Cinemre, 2004).

The Stepping Stones method involves iterating through a set of calculations to identify the shortest path between each pair of nodes in the network. The algorithm starts by selecting a node as the starting point and assigns a distance of zero. It then examines all the adjacent nodes and assigns a distance equal to the cost of the path to reach them. The algorithm then selects the node with the lowest distance and repeats the process until all nodes have been visited. The result is a matrix of distances between all pairs of nodes, which can be used to identify the shortest path between any two points.

One of the advantages of the Stepping Stones method is its ability to handle complex networks with multiple paths and constraints. For example, it can be used to optimize a transportation problem with different modes of transportation, such as air, sea, and land, and with different constraints, such as capacity limitations and time windows. By identifying the shortest path between all pairs of nodes, the Stepping Stones method can help transportation and logistics companies optimize their operations and reduce costs. Overall, the Stepping Stones method is a valuable tool for solving complex optimization problems in a wide range of industries.

2.3.2 Modified Distribution Method

The Modi Method, which was introduced with the creation of the stepping stone approach, reduced the number of solution stages and iterations, making it easier to find the ideal solution from the first solution (Roccaferrera, 1964).

The Modi Method involves three main steps. The first step is to find the initial feasible solution using the northwest corner method or the least cost method. The second step is to calculate the penalties for each empty cell in the initial feasible solution.

The penalty for an empty cell is the difference between the two lowest costs in the row and column of that cell. The third step is to identify the cell with the highest penalty and add the corresponding unit to that cell. The process is repeated until an optimal solution is obtained.

The Modi Method is an effective way to solve transportation problems because it allows for quick identification of the optimal solution. This is particularly useful in supply chain management, where time and efficiency are critical factors. The Modi Method has been used in a variety of industries, from manufacturing to healthcare, to optimize transportation and logistics. Overall, the Modi Method is a valuable tool for businesses looking to streamline their operations and reduce costs.

2.4 Special Cases of the Transportation Problem

2.4.1 Degenerate Cases

If there are *m+n-1* number of fully occupied cells in any transportation model, a closed loop line may be created for each unoccupied cell, and the variables u_i and v_j can be determined for the modi method. In other words, the optimality test can only be applied to a transportation problem with $m + n - 1$ task cells, and iterations can be repeated until the optimal solution is found. However, in any issue where the number of cells in charge is less than *m+n-1*, "deterioration," also known as "corruption" or "degeneration," is observed and must be fixed. This scenario can be encountered in two ways;

1. The scenario in which the number of existing cells is more than $m + n - 1$: Such a scenario is only faced when finding the initial basic feasible solution. Existence of the degenerate situation in this way is a precursor to an error made in the creation of the mathematical model or the application of solution techniques. As a result, the inaccuracy should be repaired by validating the model's accuracy and applying the solution technique to equate the number of filled cells to $m + n - 1$.

2. When there are fewer incumbent cells than $m + n - 1$ This kind of disruption can occur during iterations as the problem approaches optimality as well as when establishing the initial basic optimal solution. The corruption should be fixed with certain specialized procedures because it will be impossible to construct a closed loop line with an insufficient number of assigned cells or to compute the dual variables of the modi method(Levin et al., 1993).

3. $N = m + n - 1$ and $M < N$; If the number of cells that should take part is *N* and the number of cells that have taken part is *M*, the degeneration level of the solution is *"$N - M$"*(L.S.Goddard.,1963).

As stated, the second type of degeneration can occur while obtaining the initial basis feasible solution or in the iteration stages. In both methods, the tool used to eliminate the degeneration situation is the same.

It is placed in empty cells until the number of cells that need to take charge E(epsilon), which represents a small and positive value very close to zero, is $m + n - 1$. However, after this correction process, the solution stages can be continued. (Thierauf, 1978).

In transportation models, E is subject to the following hypotheses:

a. $E < xij\ , for\ each\ xij > 0$

b. $E + 0 = E$

c. $Xij\ \pm E = xij\ , for\ each\ xij > 0$

d. If there are two or more E's in the solution and E is in a row higher than E; $E < E'$. If E and E' are on the same line and E is further left, the rule $E < E$' is valid.

2.4.2 Unbalanced Cases

In order to apply the solution technique of the unbalanced transportation model to any problem, the problem must be transformed into a balanced model with a simple technique. In order to ensure the equality of aggregate supply and aggregate demand, dummy production or consumption centers with zero transportation costs should be added to the problem.

The first step in turning an unbalanced model into a balanced model is to look at the discrepancy between aggregate supply and aggregate demand. Two instances show an imbalance between total supply and total demand. Either the aggregate supply exceeds the aggregate demand, or the aggregate demand exceeds the aggregate supply.

2.4.3 Empty Cells

In the context of the transportation problem, empty cells refer to a situation where the available supply and demand do not match, leaving some cells in the transportation matrix unoccupied. These empty cells can arise due to a variety of reasons, such as incomplete information about the supply and demand, or constraints that limit the maximum amount of goods that can be transported from one location to another. It is important to handle empty cells effectively to ensure that the transportation problem can be solved optimally.

One common method of dealing with empty cells is to introduce dummy variables to balance the supply and demand. Dummy variables are fictitious locations that do not exist in reality, but allow for the transportation matrix to be balanced. For example, if there is a shortage of supply at a particular location, a dummy location can be added with a supply equal to the difference between the demand and supply. Similarly, if there is excess supply at a location, a dummy location can be added with a demand equal to the excess supply. The introduction of dummy variables allows for the transportation problem to be solved optimally, even in the presence of empty cells.

Another approach to handling empty cells is to use a heuristic method, such as the Vogel's approximation method. This method involves identifying the two smallest costs in each row and column of the transportation matrix and calculating the difference between them. The cell with the largest difference is then selected for allocation. This process is repeated until all supply and demand have been allocated. The Vogel's approximation method can be effective in solving transportation problems with empty cells, but may not always produce the optimal solution.

In conclusion, empty cells in the transportation problem can arise due to various reasons and need to be handled effectively to ensure that the problem can be solved optimally. The use of dummy variables and heuristic methods such as Vogel's approximation method are some of the approaches that can be used to handle empty cells in transportation problems.

2.4.4 Multiple Allocations

In transportation problems, multiple allocations can occur when a source has more supply than a single transportation route can handle, or when a destination has more demand than can be satisfied by a single transportation route. It's a common challenge that must be handled effectively to find the optimal transportation plan that minimizes the overall cost.

One way to address multiple allocations is by using the stepping stone method. It's a time-consuming approach but ensures the optimal solution is found. It involves identifying an empty cell in the transportation matrix, calculating the cost of allocating one unit of the shipment to that cell, and comparing it to the costs of the existing allocations along the routes that pass through the cell. If the cost of the allocation is lower, the allocation is updated, and the process is repeated for the next empty cell.

Another approach is the MODI method, which is more efficient than the stepping stone method as it involves fewer calculations. It identifies the cells with the highest opportunity cost, which represent the optimal cells for re-allocating the shipment. This method is great when you need to find the optimal solution when multiple allocations are present.

The northwest corner method is another method that is less complex than the others. It starts from the top left corner of the transportation matrix and allocates as much supply as possible to the destination along the first row. It then moves to the next row and allocates the remaining supply along the row until all supply has been allocated. This process is repeated for the remaining columns until the demand has been satisfied. It may not produce the optimal solution but can serve as a starting point for further optimization using the stepping stone or MODI method.

To sum up, multiple allocations in transportation problems require careful consideration to find the optimal transportation plan. The choice of method depends on the complexity of the transportation problem and the available resources. Whether you choose the stepping stone, MODI, or northwest corner method, the goal is to minimize the overall cost while allocating resources along multiple routes.

2.5 Transportation Problem in a Network

2.5.1 Optimal Path Problem

Finding the shortest route between two graph vertices (or nodes) is the goal of the shortest path problem. The shortest path problem is solved using algorithms like the Floyd-Warshall algorithm and several iterations of Dijkstra's algorithm. Road networks, logistics, communications, electrical design, power grid contingency analysis, and community detection are just a few examples of where the shortest path problem is applied.

We differentiate many versions of the shortest path problem:

- In the Single-Pair Shortest Path problem, we must determine the shortest path between two vertices.
- The shortest pair between two vertices must be found in the single-source shortest path problem.
- Find the shortest pair between a pair of vertices in the Single-Destination Shortest Path problem.
- The All-Pairs Shortest Path problem requires us to identify the shortest pathways between every pair of vertices in the graph.

The following are the most significant algorithms for resolving the shortest path problem:

- For unweighted graphs, a Breadth-first search identifies the shortest pathways.
- For a network with non-negative edge weights, Dijkstra's algorithm resolves the single-source shortest path problem.
- For a graph with potential negative edge weights, the Bellman-Ford algorithm resolves the single-source shortest path problem.
- The single-pair shortest path problem is solved by the A* algorithm utilizing heuristics to expedite the search.
- For a graph with potential negative edge weights, the Floyd-Warshall algorithm finds the all-pairs shortest path problem.

2.5.2 Maximal Flow Problem

The Maximum Flow Problem is a sort of transportation problem that includes finding the maximum amount of flow that may be sent across a network from a source node to a destination node. The problem occurs in a variety of real-world applications, including communication networks, water distribution, and traffic flow.

The maximum flow problem can be expressed as a network flow problem, where each network edge represents a pipe or channel with a specific flow capacity. Finding the maximum amount of flow that can be transmitted from the source node to the destination node while respecting the capacity restrictions of each edge and making sure that the flow conservation law is satisfied at each node is the objective.

The Ford-Fulkerson algorithm, the Edmonds-Karp algorithm, and the Dinic's algorithm can all be used to solve the maximum flow problem. These methods utilize different approaches to discover the maximum flow, but they all rely on the concept of augmenting paths, which are paths in the network that can transport additional flow.

The Ford-Fulkerson algorithm is a straightforward algorithm that repeatedly finds an augmenting path in the network and raises the flow along that path until no more augmenting paths can be located. After it reaches its maximum flow, the algorithm stops, but in the worst situation, it can take an exponentially long time.

The Edmonds-Karp algorithm is a Ford-Fulkerson version that does a breadth-first search to identify the shortest augmenting path in the network. This technique assures that the algorithm executes in polynomial time, namely O(V^2 E), where V is the number of nodes and E is the number of edges in the network.

The more sophisticated Dinic's algorithm employs a tiered method to discover the augmenting pathways. It generates a layered graph, where each layer represents the nodes that may be reached from the source node with a specified amount of edges. The augmenting paths are then discovered by the algorithm using a depth-first search on the layered graph. Dinic's algorithm has a temporal complexity of O(E V^2) in the worst case, but it is often faster in reality than the Edmonds-Karp algorithm.

3 Java

When the first computers were made, there were no high-level programming languages like FORTRAN, COBOL, Pascal, C/C++, or Java. Different machines have different hardware structures, so the machine languages of different brands and models of computers are also different. Because of this, the first programmers could only make the computer work by using the machine's own language. Machine language is hard to learn, and what you learn for one brand or model of machine doesn't work for another brand or model. Assembly language and then high-level languages emerged as a solution to this problem. Programmers found considerable comfort in high-level languages. Because the programmer could write the source code in any language he wanted without thinking about the operating system and the machine. For example, a C source program could be compiled and run on any machine and operating system with the right compiler(Roller, 2023).

High-level languages like FORTRAN, COBOL, Pascal, Modula, and C, which are now referred to as procedural languages, have been useful to programmers for a long time and still are. But there was a problem. A source program that was compiled on a certain kind of computer using a certain operating system could only be run on that kind of computer using that operating system. When the operating system and/or machine type changes; for example, when the platform changed, the program couldn't run there; it had to be recompiled with a compiler appropriate for the new platform. We shorten this to "platform dependency." What we mean by "platform dependency" is that a source program can only run on certain kinds of computers because it was compiled using a certain compiler and an operating system. For example, you cannot run a computer program that was written on a PC with Windows operating system on a Macintosh or Linux computer((GeeksforGeeks, 2023).

The problem has been made worse by the market's proliferation of various operating systems and hardware types, the growth of computer networks, and the fact that the devices linked to a network are made up of computers of various makes and models that use various operating systems.

A platform-independent language had to be developed in order to solve the problem. Java, a language created by Sun, was used to complete this task. The Sun firm, however, did not set out to address this significant problem; rather, they had a simpler goal in mind. He sought to create a language that would facilitate the usage of electrical equipment. He quickly saw that this task could only be completed on a shared platform.

The Java programming language, which works on any platform, was developed in 1995 by James Gosling, a programmer for Sun.

Gosling created a brilliant, straightforward invention. He created a generic virtual machine that can be set up on various hardware and operating systems. Free copies of this virtual machine, known as JVM (Java Virtual Machine), were made available. Any platform can easily install JVM. The Java compiler transforms source Java programs into a form of machine language that can be executed on the JVM. The JVM interprets and runs this program, which is known as Java Bytecode. Java bytecode serves a similar purpose to source code created and compiled in procedural languages.

The JVM can be installed on any computer by anyone who wishes to use Java programs. Downloading and installing the Java Runtime Environment (JRE) application from the internet will enough for this. The JRE is only installed once on the PC. After that, this computer may execute any Java application. The JRE automatically generates the JVM virtual machine when a Java application is executing on the system. The JVM is a program that executes as needed and, like all programs, is removed from main memory once it has served its purpose. As a result, the machine cannot be damaged. Most modern browsers may now automatically download and install the JRE when they run into Java apps((The Editors of Encyclopaedia Britannica, 2024).

Java is a straightforward, contemporary, object-oriented, type-protected language with all the positive traits of C and C++. Also, it can operate on any platform. He was able to use this talent to find a variety of uses for computers, the internet, mobile devices, game consoles, and household goods. Java can therefore be viewed as both a programming language and an environment. The operating system, networks, internet programming, databases, and all middleware technologies are all included in this ecosystem.

4 Practical Part

4.1 Application idea

The main idea of the application is the help students with their studies and lectures, the application will solve transportation problem step-by-step. The results will be shown in detail at the end of the solution. For example, when a student wants to solve a transportation problem whose dimensions are determined by him/her, he/she can create his/her own transportation table. Once created, they can fill the table with their own data and see the results in every detail. After seeing the results and explanations, students can understand the solutions more easily and increase their success in their lessons.

4.1.1 Target Audience Analysis

There are thousands of students are joining the Czech University of life science every year. These students can include many nationalities, languages, ways of life and levels of intelligence. For helping the students who needs or students which want to understand better the Transportation problem can use the application. The application will help them to understand better the transportation problem and they can succeed and get better grades in their field.

4.1.2 Application Overview

In this thesis, Java application named "Step-by-step Solver for Transportation Problems" is developed. It's designed for students, especially those studying operations research and logistics. The app makes learning about transportation problems easier. You can enter different values like costs, supply, and demand, and then choose a method to solve the problem. It's a practical tool that brings theory to life, helping you understand how to apply classroom knowledge to real-world situations. This app aims to make complex concepts more accessible and engaging for students.

4.1.3 Alternative Solutions

There some applications for solving Transportation Problem but there is a need for another applications in this field. The applications which are used for solving Transportation problem can have additional steps. They can be expensive and some of them can be hard to install. The main problem is that the user can not see the steps.

One of the applications which is used for solving Transportation Problem is Excel but solving transportation problem with excel is can be challenging sometimes. The user should make additionals settings and the user should know how to use Excel for reaching the solution. Even if the user knows how to use Excel, it takes time to get to the result and the user may encounter an error. The user will lose time trying to find that error again.

The another app is used for solving Transportation Problem is Qm for Windows. This application is better as a user experience and it is easy to use.This application is more preferable for users because it is also easy to install and also because Excel, for example, is a paid application.

If the user is a student, they can use Microsoft's 365 service, but it is more difficult to install than QM for Windows. At the same time, once installed, the user can create the table and get the solution by selecting the Transportation section from the left without the need for new settings, but the missing part of this application for the user is that it does not write how the solutions are obtained. This makes it difficult for the user to understand the solutions.

There are some another Transportation Problem solvers app and those are created for mobile infrastructure but those applications have limited abilities. In those applications user can not create bigger transportation tables and the applications don’t show how the solution is reached.

4.1.4 Requirement Analysis

SWOT analysis

Strength: Comprehensive Implementation

Strength Detail: The code includes a variety of methods for solving transportation problems, such as the North West Corner Method, Least Cost Method, Vogel's Approximation Method, Russell's Approximation Method, MODI Method, and Stepping Stone Method. This diverse range of options demonstrates an extensive understanding of the domain, providing flexibility for different problem scenarios.

Weakness: Lack of Modularity

Weakness Detail: The code is written as a single, monolithic structure, lacking modular design. This approach makes maintenance, updates, and extensions challenging, and reduces readability and accessibility for new developers or educators.

Opportunity: Refactoring for Modularity

Opportunity Detail: The current structure of the code offers an opportunity for significant improvement through refactoring. By breaking down the code into modules or classes, it can be made more maintainable, extendable, and easier to test and debug (Author, Year).

Threat: Technological Advancements

Threat Detail: The field of operations research and logistics is rapidly evolving, with new algorithms and methods being developed continually. If the code does not evolve with these advancements, it risks becoming obsolete and less relevant in both practical and educational applications.

Use Case Analysis

Use Case Title: Streamlining Warehouse Distribution

Primary Actor: Distribution Manager

Stakeholders and Interests:

- Distribution Manager: Aims to optimize warehouse distribution routes to minimize costs and improve efficiency.
- Warehouse Staff: Require a clear and efficient distribution plan to streamline their workflow.
- Retail Outlets: Expect timely and cost-efficient delivery of products.
- Customers: Benefit indirectly from efficient distribution through faster product availability and potential cost savings.
-

Preconditions:

- The Distribution Manager has functional knowledge of the software.
- Data regarding warehouse locations, retail outlet locations, and transportation costs are ready and accurate.

Main Success Scenario:

1. Entering Data: The Distribution Manager inputs data about warehouse locations, retail outlets, and associated transportation costs.
2. Selecting Optimization Technique: The Manager chooses a suitable optimization technique, like the Vogel's Approximation Method, based on specific distribution needs.
3. Computing Distribution Plan: The software calculates the most cost-effective and efficient distribution routes and strategies.
4. Analyzing Proposed Plan: The Manager examines the proposed distribution plan for feasibility and practicality.
5. Plan Execution: The distribution plan is rolled out for implementation in actual warehouse-to-retail outlet deliveries.
6. Monitoring and Evaluating: The effectiveness of the implemented plan is monitored and evaluated for future adjustments.

Extensions:

- Data Errors: If any errors are found in the input data, the software alerts the Manager for immediate correction.
- Adjusting to Seasonal Variations: The software can accommodate different scenarios, like seasonal demand spikes, by adjusting the distribution plan accordingly.

Special Requirements:

- The system should be capable of handling multiple warehouse and outlet data simultaneously.
- The software must provide an intuitive and informative graphical representation of distribution routes.

Technology and Data Variations:

- Compatibility with various data formats for easy integration with existing warehouse management systems.
- Mobile accessibility for on-the-go monitoring by the Distribution Manager.

Frequency of Use: Used extensively during strategic planning phases and periodically for route optimization.

Open Issues:

- Incorporating real-time traffic data to dynamically adjust distribution routes.
- Expansion of the system to integrate with global positioning systems (GPS) for real-time tracking of shipments.

Conclusion:

This use case outlines how the transportation problem-solving software can be employed to enhance the efficiency of warehouse distribution systems. It underscores the need for a versatile system capable of adapting to different data inputs and scenarios. Future enhancements could include real-time data integration and expanded tracking capabilities to further streamline distribution processes.

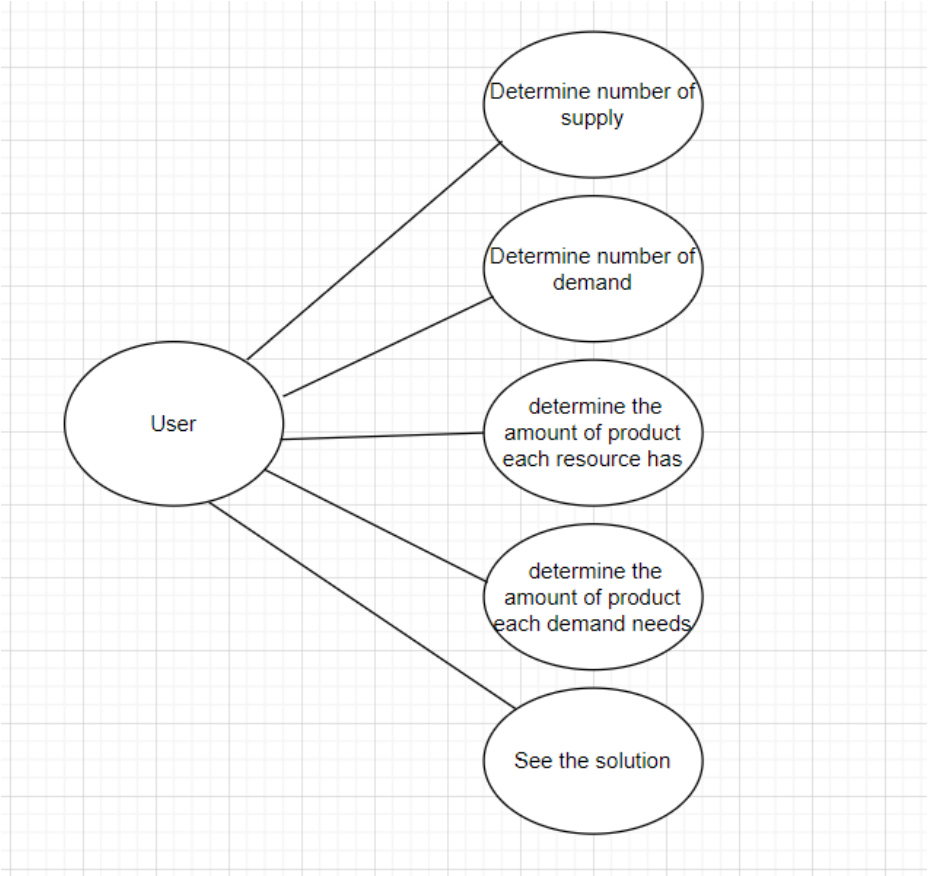


Image 3: Step-by-Step Transportation Solver Use Case Model(Source: Own material)

4.2 Implementation

4.2.1 IntelliJ IDEA Installation

To begin building the Transportation Problem-Solving application, the IntelliJ IDEA Community Edition was chosen for its straightforward setup and solid support for Java. The Community Edition is free and can be downloaded from the JetBrains website. After the download, the installation process is initiated by running the downloaded file.

The installer provides clear instructions, guiding through the steps of installation. You can accept the default configurations which are suitable for most Java projects. Once installed, opening IntelliJ IDEA will prompt you to configure the initial settings or import existing projects if you have any.

4.2.2 IntelliJ IDEA Configuration

When IntelliJ IDEA starts for the first time, it checks to ensure the Java Development Kit (JDK) is installed. The JDK is necessary for Java development, and if it isn't found, the IDE provides instructions for downloading and installing it.

For general use, the default settings of IntelliJ IDEA work well. However, should you need to handle larger projects with more complex calculations, you might consider increasing the memory allocation. This is done by adjusting the VM options in IntelliJ IDEA:

1. From the main menu, select 'Help'.
2. Choose 'Edit Custom VM Options'.
3. If prompted, confirm you want to create a custom VM options file.
4. Once the file is open, you can change the -Xmx value to increase the maximum heap size.

4.2.3 Starting a New IntelliJ IDEA Project

Creating a new project in IntelliJ IDEA is straightforward:

1. Click on 'File' in the menu bar, then 'New', and select 'Project'.
2. In the New Project window, make sure 'Java' is selected on the left side.
3. Choose a project SDK (IntelliJ should automatically detect the installed JDK).
4. Click 'Next', then 'Finish' after naming your project, such as 'Transportation'.

The project window will show the src folder where you can create new Java classes. To add a new class:

1. Right-click on the src folder.
2. Select 'New', then 'Java Class'.
3. Name your class, for example, 'Main'.

This Main class will be where you write the methods to solve transportation problems. You can input data, invoke solving methods, and print out results in this class.

To run the application, right-click on the Main class in the Project Explorer and select 'Run Main.main()'. The IntelliJ IDEA will compile and execute your Java code, displaying the results in the built-in console at the bottom of the IDE.

IntelliJ IDEA makes Java development accessible and less intimidating for beginners. With its integrated tools and user-friendly environment, you can focus on solving the transportation problem without worrying too much about the complexities of the development setup.

4.2.4 Implementation of Project

Transportation Class

The Transportation class, a pivotal component of a Java program, is designed to address a classic problem in logistics: optimizing the distribution of goods from multiple supply points to various demand points. This document aims to provide a comprehensive breakdown of the class, making it accessible to students and novice programmers.

Class Structure and Its Components

At its core, the ‘Transportation’ class revolves around solving the transportation problem using various methods, with a focus on user interaction and algorithmic implementation.

Graphical User Interface (GUI)

The class employs Swing, a robust Java library for creating graphical user interfaces. The GUI's design is user-centric, focusing on ease of use and clarity.

Window Setup

- Initialization: The method createAndShowGUI() kickstarts the GUI process. It initializes a JFrame - the main window titled "Transportation Problem".
- Window Specifications: The frame is set to a size of 1400x750 pixels, providing ample space for inputs and visualization. The default close operation is set to EXIT_ON_CLOSE, ensuring the application terminates correctly when the user closes the window.

Layout and Components

- Panel Layout: A JPanel named mainPanel is created and added to the frame. It uses a null layout, meaning components are positioned absolutely, giving precise control over their placement.
- Label Addition: Labels like "Number of Suppliers" and "Number of Demands" are added. They guide the user on where to input respective data.
- Text Field Dynamics: The methods addMultipleTextFields1() and addMultipleTextFields2() dynamically generate text fields within the main panel. These fields are where users input supply, demand, and cost data.

Interactivity and Data Retrieval

- Buttons for Action: Two buttons, "Set Input Area" and "Solve the Problem", are added. The 'Set Input Area' button triggers the creation or adjustment of input fields based on the user's specification of supply and demand points. The 'Solve the Problem' button initiates the problem-solving process.

- Event Handling: The application listens for actions like button clicks. For instance, upon clicking 'Set Input Area', the program fetches values from the relevant text fields and adjusts the GUI accordingly.

User Experience

- User Input Facilitation: The GUI is designed to be intuitive, with clearly marked areas for entering data. Error messages are displayed through dialog boxes to guide the user in case of invalid inputs.
- Visual Feedback: As the user interacts with the GUI, immediate visual feedback, such as the appearance of text fields and error dialogs, keeps the user informed and engaged.

Data Collection

The class employs a user-friendly Graphical User Interface (GUI) for data input, which is a cornerstone for its functionality.

GUI - The Gateway to Data Collection

- Intuitive Interface Design: The class leverages Swing components to construct an interface that is both intuitive and responsive. The use of JTextField arrays (textFields1 and textFields2) is particularly noteworthy. These text fields serve as the primary means for the user to input data, dynamically adjusting to the size of the transportation problem based on user input.
- Dynamism in GUI Elements: The dynamic creation of these text fields through methods like addMultipleTextFields1() and addMultipleTextFields2() is an elegant solution to cater to varying problem sizes. This adaptability enhances user experience and ensures that the GUI can handle a range of transportation problems.

The Art of Data Collection

- Initial Setup - Defining the Problem Size: The initial user inputs, namely the number of supply and demand points, are pivotal. They determine the scale of the data entry grid, setting up the structure for the detailed input that follows.
- Detailed Data Input - Crafting the Problem: Post the initial setup, users enter specific data – the costs associated with transportation, and the quantities of supply and demand. This detailed data is the backbone of the transportation problem to be solved.

Ensuring Data Integrity

- Robust Input Validation: The class doesn't just passively collect data; it actively validates it. This validation includes checks for numeric values, adherence to predefined constraints (like maximum points), and logical consistency in the data provided.
- Error Messaging - Guiding the User: The application employs informative error messages to guide users through correct data entry. This feature is crucial in enhancing user experience and ensuring the reliability of the data collected.

Data Processing - The Core of the Application

- Data as the Driving Force: The collected data is the fuel for the class's algorithms. These include methods like the North West Corner Method, Least Cost Method, Vogel's Approximation Method, and others. Each piece of data directly influences the behavior and outcome of these algorithms.
- Adaptive Algorithms: A notable aspect of these algorithms is their adaptability. They adjust their computational paths based on the input data, ensuring that the solution is tailored to the specific problem posed by the user.

Beyond Data Collection

- Integration with Algorithms: The seamless integration of the GUI for data collection with the computational logic of the algorithms is a hallmark of the Transportation class. This integration ensures that the transition from data input to problem-solving is smooth and efficient.
- Visual Feedback and Results: Post computation, the application provides visual feedback to the user. This feedback includes not just the solution to the transportation problem but also insights into the workings of the algorithms used.

Solving The Problem

The solveProblem method in the Transportation class is a comprehensive function designed to address the transportation problem using a variety of problem-solving algorithms. Each algorithm has its unique approach to finding the most cost-effective way to distribute goods from several suppliers to numerous consumers while adhering to supply and demand constraints. This detailed exploration will cover key methodologies including the North West Corner method, Least Cost method, Vogel's Approximation method, and others, focusing on their implementation and significance in solving the transportation problem.

North West Corner Method

The North West Corner method serves as a straightforward, initial feasible solution approach. This method starts at the north-west corner of the cost matrix (top left) and allocates supplies to demands in a manner that moves either down or right in the matrix, hence the name. It emphasizes simplicity and speed over achieving the least cost, making it an excellent starting point for iterative improvement methods.

In the solveProblem method, the implementation begins by iterating over the matrix, assigning the minimum of supply and demand to the current position. The allocations continue until either the supply or demand is exhausted. If a row's supply is depleted, the method moves to the next row (downwards); if a column's demand is satisfied, it proceeds to the next column (rightwards). This process repeats until all supplies and demands are allocated. Although not guaranteed to be optimal, it provides a baseline for comparison with more sophisticated methods.

Least Cost Method

The Least Cost method improves upon the North West Corner by considering the costs associated with each transportation route. It selects the route with the lowest cost for allocation first, aiming to minimize the overall cost from the outset. This method iterates through the cost matrix to identify the minimum cost cell that hasn't been allocated yet. The allocation is the minimum of the corresponding supply or demand, after which the method zeroes out either the row or column to prevent further allocation to that route. This strategy is more refined than the North West Corner method, as it directly targets cost reduction from the initial solution. However, it doesn't always result in the optimal solution due to its 'greedy' nature—making local optimum choices at each step without considering the overall problem structure.

Vogel's Approximation Method (VAM)

Vogel's Approximation Method offers a more sophisticated approach by considering the penalty of not using the cheapest route. It calculates the difference between

the two lowest costs in each row and column, known as the penalty, and prioritizes the allocation for the row or column with the highest penalty. This method aims to minimize the regret of not choosing the least expensive option, thereby providing a closer approximation to the optimal solution.

VAM's implementation in the “solveProblem” method involves a loop that calculates penalties, identifies the row or column with the highest penalty, and then allocates supply or demand based on the least cost in that row or column. This process is repeated until all allocations are complete. Vogel's method is particularly effective for providing a good initial solution that can be further optimized or used as a standalone solution in many practical scenarios.

Other Methods

The “solveProblem” function also introduces other sophisticated methods such as the Modi Method and the Stepping Stone Method for optimizing the initial solutions provided by the aforementioned algorithms. These methods iteratively adjust the solution by examining potential improvements along closed loops or paths within the cost matrix, ensuring that each move towards optimization reduces the overall cost.

The Modi Method uses the concept of opportunity cost to identify less costly routes, while the Stepping Stone Method systematically searches for potential cost-saving routes through an iterative process. Both methods require a feasible initial solution, which can be provided by any of the initial solution methods discussed above.

4.3 Parts of the Coding

Packages and Imports

The application is encapsulated within the transportation package, serving as a unique namespace that facilitates the organization of classes and prevents naming conflicts with other parts of the program or external libraries.

To support its functionality, the program imports several Java packages:

javax.swing.*: This package is essential for creating the graphical user interface (GUI), including elements like windows, buttons, labels, and text fields, providing the user with a visual and interactive experience.

java.util.ArrayList: The ArrayList class is utilized for its dynamic array capabilities, allowing for the flexible storage and management of JTextField components that users interact with.

java.io.*: This package includes classes necessary for input and output operations, such as FileWriter for writing to files and FileReader and BufferedReader for reading from files, facilitating data persistence and retrieval.

java.awt.*: The Abstract Window Toolkit (AWT) package provides classes for more foundational GUI components and desktop integration, such as opening files with the system's default applications.

static java.lang.Math.min: The static import of the min method is a convenience for mathematical operations, allowing for the straightforward determination of the minimum between two numbers, a frequently needed calculation in optimization routines.

The Transportation Class

Central to the application is the Transportation class, which contains all the logic and infrastructure required to solve the transportation problem.

Fields

The class defines several static variables and arrays to hold GUI components, such as JTextField instances, and numerical data representing supply and demand points, costs, and the arrays used for computing the solution.

Main Method

The main method acts as the application's entry point. It initializes the GUI within the event dispatch thread, a critical step for ensuring that the application is thread-safe and operates smoothly.

GUI Methods

createAndShowGUI(): This method sets up the main window and its components, including the frame, panels, labels, and buttons. It also handles the assignment of action listeners to buttons, enabling user interaction with the application.
addMultipleTextFields1() and addMultipleTextFields2(): These methods are responsible for dynamically adding text fields to the GUI. These fields allow users to input data related to supply and demand points, which are vital for formulating the transportation problem.

Problem Solving Methods

solveProblem(): This method orchestrates the resolution of the transportation problem. It involves initializing data structures with user input, applying various solution strategies, and presenting the results.

fillInitialArrays(): Prepares the arrays with supply, demand, and cost data as entered by the user, setting the stage for the application of solution algorithms.

printTable(): A utility for formatting the current state of the problem and the solutions into a string, which can be logged or displayed to the user.

Solution Algorithms

The class implements a variety of methods to solve the transportation problem, each employing a distinct heuristic or optimization approach. These methods include the north-west corner method, least cost method, Vogel's approximation method, Russell's approximation method, and others, all aimed at efficiently distributing supplies to demands at minimum cost.

Utility Methods

writeToFile(): Saves the detailed solution and any relevant logs to a file, providing a permanent record of the application's output.
dummyCheck(): Ensures the total supply matches total demand by potentially adding dummy rows or columns, a necessary adjustment for certain solution techniques.
sumProduct(): Calculates the total cost of the transportation plan, a key outcome of the problem-solving process.
GUI Components
The application makes extensive use of GUI components to create an interactive user experience. These include:

JFrame: The main window where the application's GUI is displayed.
JPanel: Containers within the main window that organize other GUI components.
JLabel: Text labels that provide information to the user.
JTextField: Fields where users can enter data.
JButton: Buttons that users can click to trigger specific actions, such as solving the problem or adding input fields.

4.3.1 Example Solution with Transportation Solver App

1. First we open the application step by step solver. In the main page we can write number of supplies and number of demands

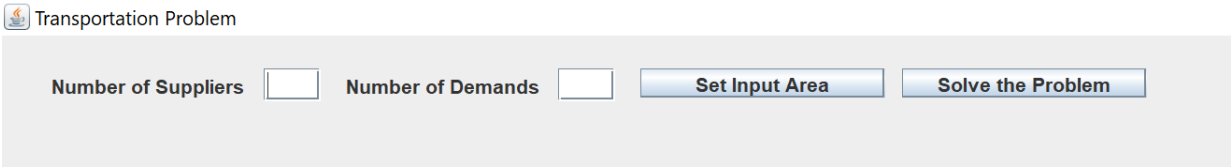


Image 4: Entrance page of step-by-step solver(Source: Own Source)

2. For the example 2 supplies and 2 demands were chosen. The spaces were written according to chosen numbers. After the right click to the button Set Input Arena the user has got 2x2 size transportation table.

Image 5: Creation of 2x2 Transportation table in the application(Source: own source)

3. The User fullfill the table according the his choices when the user right clicked Solve the Problem button there will be shown a message and message will be "Problem solved. "

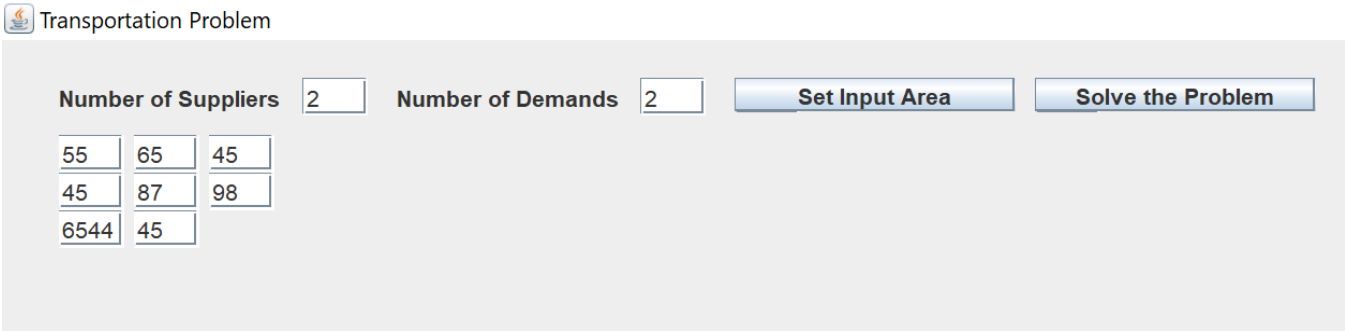


Image 6: Filling the transportation table in the application(Source: own source)

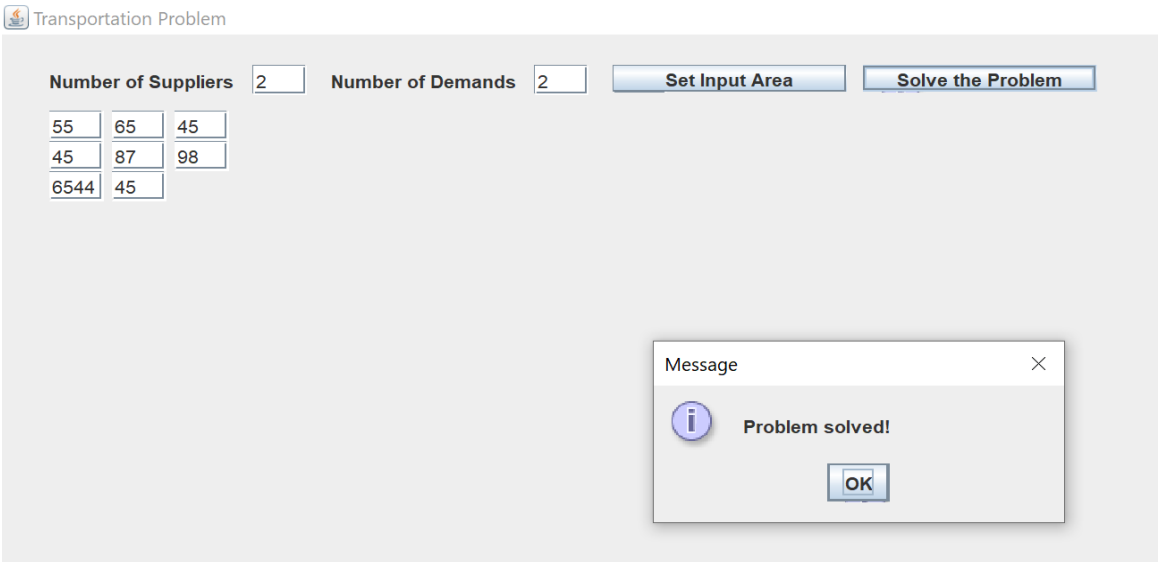


Image 7:Problem solved message in the Step-by-Step solver of application.(Source:own Source)

4. In the final step OK button will be clicked by user and the solution.txt will be opened. In this txt file solution will be shown.

solution.txt - Notepad

File Edit Format View Help

```
 ----------------------------------
|     NORTHWEST CORNER METHOD      |
 ----------------------------------

The initial problem is

45              68              98
650             245             65
78              98

Total supply is 163 unit and total demand is 176 unit
Thus, total supply < total demand. We add dummy supply point!

45              68              98
650             245             65
0               0               13
78              98

Supply point 1 has 98 unit free capacity and demand point 1 has 78 unit unsatisfied demand
Minimum is 78
This meets the complete demand of demand point 1 and leaves 20 unit supply of supply point 1

45(78)          68              20
650             245             65
0               0               13
0               98

We go horizontally!
Supply point 1 has 20 unit free capacity and demand point 2 has 98 unit unsatisfied demand
Minimum is 20
This exhausts the capacity of supply point 1 and leaves 78 unit demand of demand point 2

45(78)          68(20)          0
650             245             65
0               0               13
0               78

We go vertically!
Supply point 2 has 65 unit free capacity and demand point 2 has 78 unit unsatisfied demand
Minimum is 65
This exhausts the capacity of supply point 2 and leaves 13 unit demand of demand point 2

45(78)          68(20)          0
650             245(65)         0
0               0               13
0               13

We go vertically!
Supply point 3 has 13 unit free capacity and demand point 2 has 13 unit unsatisfied demand
Minimum is 13
This exhausts the capacity of supply point 3 and meets the complete demand of demand point 2

45(78)          68(20)          0
650             245(65)         0
0               0(13)           0
0               0

Solution with Northwest Corner Method is 20795
```

Image 8: Northwest Corner Method solution in the application(Source: Own Source)

solution.txt - Notepad

File Edit Format View Help

```
 ----------------------------------
|        LEAST COST METHOD          |
 ----------------------------------

The initial problem is


45              68              98
650             245             65
78              98

Total supply is 163 unit and total demand is 176 unit
Thus, total supply < total demand. We add dummy supply point!

45              68              98
650             245             65
0               0               13
78              98

Min cost is 0 at supply point 3 and demand point 1 with maximum allocation amount 13
This exhausts the capacity of supply point 3 and leaves 65 unit demand of demand point 1

45              68              98
650             245             65
0(13)           0               0
65              98

Min cost is 45 at supply point 1 and demand point 1 with maximum allocation amount 65
This meets the complete demand of demand point 1 and leaves 33 unit supply of supply point 1

45(65)          68              33
650             245             65
0(13)           0               0
0               98

Min cost is 68 at supply point 1 and demand point 2 with maximum allocation amount 33
This exhausts the capacity of supply point 1 and leaves 65 unit demand of demand point 2

45(65)          68(33)          0
650             245             65
0(13)           0               0
0               65

Min cost is 245 at supply point 2 and demand point 2 with maximum allocation amount 65
This exhausts the capacity of supply point 2 and meets the complete demand of demand point 2

45(65)          68(33)          0
650             245(65)         0
0(13)           0               0
0               0

Solution with Least Cost Method is 21094
```

Image 9: Solution of Least Cost Method in the application(Source: own source)

solution.txt - Notepad

File Edit Format View Help

```
 ----------------------------------
|  VOGEL'S APPROXIMATION METHOD    |
 ----------------------------------

The initial problem is

45              68              98
650             245             65
78              98

Total supply is 163 unit and total demand is 176 unit
Thus, total supply < total demand. We add dummy supply point!

45              68              98
650             245             65
0               0               13
78              98

Max penalty is 405 in row 2
Min cost is 245 at supply point 2 and demand point 2 with maximum allocation amount 65
This exhausts the capacity of supply point 2 and leaves 33 unit demand of demand point 2

45              68              98              23
650             245(65)         0               405
0               0               13              0
78              33
45              68

Max penalty is 68 in column 2
Min cost is 0 at supply point 3 and demand point 2 with maximum allocation amount 13
This exhausts the capacity of supply point 3 and leaves 20 unit demand of demand point 2

45              68              98              23
650             245(65)         0               -1
0               0(13)           0               0
78              20
45              68

Max penalty is 68 in column 2
Min cost is 68 at supply point 1 and demand point 2 with maximum allocation amount 20
This meets the complete demand of demand point 2 and leaves 78 unit supply of supply point 1

45              68(20)          78              23
650             245(65)         0               -1
0               0(13)           0               -1
78              0
45              68

Max penalty is 45 in row 1
Min cost is 45 at supply point 1 and demand point 1 with maximum allocation amount 78
This exhausts the capacity of supply point 1 and meets the complete demand of demand point 1

45(78)          68(20)          0               45
650             245(65)         0               -1
0               0(13)           0               -1
0               0
45              -1

Solution with Vogel's Approximation Method is 20795
```

Image 10: Solution of Vogel's Approximation Method in the application

solution.txt - Notepad

File Edit Format View Help

```
45              68              98
650             245             65
78              98

Total supply is 163 unit and total demand is 176 unit
Thus, total supply < total demand. We add dummy supply point!

45              68              98
650             245             65
0               0               13
78              98

Delta(1,1)=c(1,1)-(U(1)+V(1)=45-(68+650)=-673
Delta(1,2)=c(1,2)-(U(1)+V(2)=68-(68+245)=-245
Delta(2,1)=c(2,1)-(U(2)+V(1)=650-(650+650)=-650
Delta(2,2)=c(2,2)-(U(2)+V(2)=245-(650+245)=-650
Delta(3,1)=c(3,1)-(U(3)+V(1)=0-(0+650)=-650
Delta(3,2)=c(3,2)-(U(3)+V(2)=0-(0+245)=-245
Min delta is -673 at supply point 1 and demand point 1 with maximum allocation amount 78
This meets the complete demand of demand point 1 and leaves 20 unit supply of supply point 1

45(78)          68              20              68
650             245             65              650
0               0               13              0
0               98
650             245

Delta(1,2)=c(1,2)-(U(1)+V(2)=68-(68+245)=-245
Delta(2,2)=c(2,2)-(U(2)+V(2)=245-(245+245)=-245
Delta(3,2)=c(3,2)-(U(3)+V(2)=0-(0+245)=-245
Min delta is -245 at supply point 1 and demand point 2 with maximum allocation amount 20
This exhausts the capacity of supply point 1 and leaves 78 unit demand of demand point 2

45(78)          68(20)          0               68
650             245             65              245
0               0               13              0
0               78
-1              245

Delta(2,2)=c(2,2)-(U(2)+V(2)=245-(245+245)=-245
Delta(3,2)=c(3,2)-(U(3)+V(2)=0-(0+245)=-245
Min delta is -245 at supply point 2 and demand point 2 with maximum allocation amount 65
This exhausts the capacity of supply point 2 and leaves 13 unit demand of demand point 2

45(78)          68(20)          0               -1
650             245(65)         0               245
0               0               13              0
0               13
-1              245

Delta(3,2)=c(3,2)-(U(3)+V(2)=0-(0+0)=0
Min delta is 0 at supply point 3 and demand point 2 with maximum allocation amount 13
This exhausts the capacity of supply point 3 and meets the complete demand of demand point 2

45(78)          68(20)          0               -1
650             245(65)         0               -1
0               0(13)           0               0
0               0
-1              0

Solution with Russell's Approximation Method is 20795
```

Image 11: Solution of Russels's Approximation method in the application.(Source: Own Source)

5 Conclusion

This thesis has embarked on an exploration of the transportation problem, a staple in the field of operations research and optimization, through both theoretical frameworks and practical application. The culmination of this investigation is embodied in a Java application designed to not only illustrate the complexities inherent in the transportation problem but also to offer a tangible solution through computational means.

The theoretical foundation of the thesis illuminated the historical evolution and mathematical formulations of the transportation problem. By delving into methodologies such as the North-West Corner Method, Least Cost Method, Vogel's Approximation Method, and others, it offered a comprehensive overview of the strategies devised to approach the problem efficiently. This exploration not only highlighted the complexity of achieving optimal solutions but also underscored the significance of such models in practical, cost-sensitive decision-making processes.

It suggests enriching the theoretical foundation of the transportation problem by weaving in advanced technologies such as artificial intelligence and machine learning. The integration of these technologies is seen as a pivotal advancement, potentially transforming the conventional methodologies used to solve transportation problems.. This approach significantly bolsters the relevance and applicability of transportation models to real-world scenarios.

Moreover, the utility of the Java-based application developed as part of this thesis, while invaluable as an educational resource, presents opportunities for further refinement to enhance its practical application. Future development could focus on incorporating more sophisticated algorithms that adjust to variations in supply and demand, thereby offering a more nuanced tool for problem-solving. Additionally, improvements in user interface design and feedback mechanisms would increase efficency and enrich the user experience. Through these proposed improvements, the application could serve as a bridge between academic learning and practical application, offering a deeper, more nuanced understanding of the transportation problem to students and individuals.

In conclusion, while this thesis and its associated application mark a significant step towards making the transportation problem more accessible, they also open avenues for future research and development. By marrying theoretical insights with practical applications and embracing the potential of emerging technologies, the study contributes to the ongoing dialogue in operations research and lays the groundwork for future innovations in the field.

References

Cinemre, N. (2004). *YÖNEYLEM ARAŞTIRMASI* (2nd ed.). Beta Basim.

Dantzig, G. (2016). *Linear programming and extensions*. Princeton University Press.

GeeksforGeeks. (2023, October 31). *Generation of programming languages*. https://www.geeksforgeeks.org/generation-programming-languages/

Goddard, L. S. (1964). *Mathematical Techniques of Operational Research*. Southwestern Publishing Company.

Ignizio, J. P. (1983). Generalized goal programming An overview. *Computers & Operations Research*, *10*(4), 277–289. https://doi.org/10.1016/0305-0548(83)90003-5

Kara, I. (1991). *Doğrusal proğramlama*. Bilim Teknik Yayınevi.

Levin, R. I., & Kirkpatrick, C. (1965). *Quantitative approaches to management*. McGraw-Hill Book Company, Inc.

Özkan, Ş. (2005). *YÖNEYLEM ARAŞTIRMASI NİCEL KARAR TEKNİKLERİ*. Nobel Akademik Yayıncılık.

Phillips, D. T., Ravindran, A., & Solberg, J. J. (1976). *Operations research: Principles and Practice*. John Wiley & Sons.

Render, B., Stair, R. M., Jr., Hanna, M. E., & Badri, T. N. (2016). *Quantitative analysis for management*. Pearson Education India.

Roccaferrera, D., & Ferrero, G. M. (1964). *Operations research models for business and industry*. http://ci.nii.ac.jp/ncid/BA05571421

Roller, J. (2023, July 11). *Coding From 1849 to 2022: a Guide to The Timeline of Programming Languages*. IEEE Computer Society.

https://www.computer.org/publications/tech-news/insider-membership-news/timeline-of-programming-languages

The Editors of Encyclopaedia Britannica. (2024, January 9). *Java | Definition & Facts*. Encyclopedia Britannica. https://www.britannica.com/technology/Java-computer-programming-language

Tulunay, Y. (1987). *Matematik programlama ve işletme uygulamaları* (3rd ed.). Bayrak Matbaacılık.

Yüksel, D., Kızılay, D., Öztop, H., & Özkan, S. D. (2021). Mathematical models for milk dispatching problem. *Journal of Transportation and Logistics*, *6*(2), 217–235. https://doi.org/10.26650/jtl.2021.940506

Appendix

Source Code:

```
package transportation;

import javax.swing.*;
import java.util.ArrayList;
import java.io.FileWriter;
import java.io.IOException;
import static java.lang.Math.min;
import java.io.BufferedReader;
import java.io.FileReader;
import java.awt.*;
import java.io.File;

public class Transportation
{

    private static ArrayList<JTextField> textFields1 = new ArrayList<>();
    private static ArrayList<JTextField> textFields2 = new ArrayList<>();

    static int supplyPoints;
    static int demandPoints;
    static int numberOfRows;
    static int numberOfColumns;

    static int[] supply = new int[40];
    static int[] demand = new int[40];
    static int[] rowPenalty = new int[40];
    static int[] colPenalty = new int[40];

    static int[][] cost = new int[40][40];
    static int[][] problemArray = new int[41][41];
    static int[][] solutionArray = new int[40][40];

    static String allMessage = "";

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(() -> createAndShowGUI());
    }

    private static void createAndShowGUI()
    {
        JFrame frame = new JFrame("Transportation Problem");
        JPanel mainPanel = new JPanel();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(1400, 750);

        // Set layout for the mainPanel
        mainPanel.setLayout(null); // For simplicity, using absolute
positioning
```

```
        // Add the mainPanel to the frame
        frame.add(mainPanel);
        frame.setVisible(true);

        ///start from here

        // Add a new label to the mainPanel
        JLabel newLabel = new JLabel("Number of Suppliers");
        newLabel.setBounds(30, 15, 200, 30); // Set the position and size
of the label
        mainPanel.add(newLabel);

        // Add a new label to the mainPanel
        JLabel newLabe2 = new JLabel("Number of Demands");
        newLabe2.setBounds(210, 15, 200, 30); // Set the position and
size of the label
        mainPanel.add(newLabe2);

        frame.setVisible(true);
        frame.setVisible(true);

        //end here
        // Call the addTextField method to add the JTextField dynamically
        addMultipleTextFields1(mainPanel);

        JButton retrieveButton = new JButton("Set Input Area");
        retrieveButton.setBounds(390, 20, 150, 18);
        mainPanel.add(retrieveButton);

        JButton solveButton = new JButton("Solve the Problem");
        solveButton.setBounds(550, 20, 150, 18);
        mainPanel.add(solveButton);

        retrieveButton.addActionListener(e ->
        {
            JTextField textFieldToGetValue1 = textFields1.get(0);
            JTextField textFieldToGetValue2 = textFields1.get(1);

            if (textFieldToGetValue1.getText().equals(""))
            {
                JOptionPane.showMessageDialog(null, "Enter a valid value
for supply points!");
            }
            else if (textFieldToGetValue2.getText().equals(""))
            {
                JOptionPane.showMessageDialog(null, "Enter a valid value
for demand points!");
            }
            else if (Integer.parseInt(textFieldToGetValue1.getText())>40
|| Integer.parseInt(textFieldToGetValue2.getText())>40)
            {
                JOptionPane.showMessageDialog(null, "Allowed problem size
is exceeded!");
            }
            else
            {
```

```
                supplyPoints =
Integer.parseInt(textFieldToGetValue1.getText());
                demandPoints =
Integer.parseInt(textFieldToGetValue2.getText());

                int listSize = textFields2.size();

                if (listSize!=0)
                {
                    for (int i=0; i<listSize;i++)
                    {
                        JTextField textFieldToRemove =
textFields2.get(i); // Remove the first textField from the list
                        mainPanel.remove(textFieldToRemove); // Remove
the textField from the panel
                    }
                    mainPanel.revalidate();
                    mainPanel.repaint();
                    textFields2.clear();

addMultipleTextFields2(mainPanel,(supplyPoints+1),(demandPoints+1), 30,
50);
                }
                else
                {

addMultipleTextFields2(mainPanel,(supplyPoints+1),(demandPoints+1), 30,
50);
                }
            }
        });

        solveButton.addActionListener(e ->
        {
            int listSize = textFields2.size();
            if (listSize==0)
            {
                JOptionPane.showMessageDialog(null, "Please create a
input table!");
            }
            else
            {
                int myCheck=0;
                for(int i=0;i<listSize;i++)
                {
                    JTextField textFieldToGetValue = textFields2.get(i);
                    if (textFieldToGetValue.getText().equals(""))
                    {
                        myCheck = 1;
                        break;
                    }
                }

                if (myCheck==1)
                {
```

```
                    JOptionPane.showMessageDialog(null, "Please fill
entire input table!");
                }
                else
                {
                    solveProblem();
                }
            }
        });
    }

    public static void addMultipleTextFields1(JPanel panel)
    {
        int x = 160;
        int y = 20;
        int textFieldWidth = 35;
        int textFieldHeight = 20;
        int horizontalSpacing = 180;

        for (int i = 0; i < 2; i++)
        {
            JTextField textField = new JTextField();
            textField.setBounds(x, y, textFieldWidth, textFieldHeight);
            textFields1.add(textField);
            panel.add(textField);
            x += horizontalSpacing; // Increment y-coordinate for the
next textField
        }
        panel.revalidate();
        panel.repaint();
    }

    public static void addMultipleTextFields2(JPanel panel, int count1,
int count2, int x, int y)
    {
        int xx = x;
        int yy = y;
        int textFieldWidth = 35;
        int textFieldHeight = 20;
        int verticalSpacing = 20;
        int horizontalSpacing = 40;


        for (int i = 1; i <= count1; i++) {
            for (int j = 1; j <= count2; j++){
                if (i==count1 && j==count2)
                    continue;

                JTextField textField = new JTextField();
                textField.setBounds(xx, yy, textFieldWidth,
textFieldHeight);
                textFields2.add(textField);
                panel.add(textField);
                xx += horizontalSpacing;
            }
            yy += verticalSpacing;
```

```
            xx=x;
        }
        panel.revalidate();
        panel.repaint();
    }

    public static void solveProblem()
    {
        fillInitialArrays();

        allMessage = " ----------------------------------\n";
        allMessage+= "|     NORTHWEST CORNER METHOD      |\n";
        allMessage+= " ----------------------------------\n";
        allMessage+= "\nThe initial problem is\n";

        printTable();
        dummyCheck();
        printTable();
        northWestCornerMethod();

        fillInitialArrays();

        allMessage +="\n\n\n";
        allMessage += " ----------------------------------\n";
        allMessage += "|        LEAST COST METHOD          |\n";
        allMessage += " ----------------------------------\n";
        allMessage += "\nThe initial problem is\n\n";

        printTable();
        dummyCheck();
        printTable();
        leastCostMethod();

        fillInitialArrays();

        allMessage +="\n\n\n";
        allMessage+= " ----------------------------------\n";
        allMessage+= "|  VOGEL'S APPROXIMATION METHOD    |\n";
        allMessage+= " ----------------------------------\n";
        allMessage+= "\nThe initial problem is\n";

        printTable();
        dummyCheck();
        printTable();
        vogelsApproximationMethod();

        fillInitialArrays();

        allMessage +="\n\n\n";
        allMessage+= " ----------------------------------\n";
        allMessage+= "|  RUSSELL'S APPROXIMATION METHOD  |\n";
        allMessage+= " ----------------------------------\n";
        allMessage+= "\nThe initial problem is\n";

        printTable();
        dummyCheck();
```

```
        printTable();
        russellsApproximationMethod();

        /*fillInitialArrays();

        allMessage +="\n\n\n";
        allMessage+= " ----------------------------------\n";
        allMessage+= "|             MODI METHOD           |\n";
        allMessage+= " ----------------------------------\n";
        allMessage+= "\nThe initial problem is\n";

        printTable();
        dummyCheck();
        printTable();
        modiMethod();

        fillInitialArrays();

        allMessage +="\n\n\n";
        allMessage+= " ----------------------------------\n";
        allMessage+= "|      STEPPING STONE METHOD        |\n";
        allMessage+= " ----------------------------------\n";
        allMessage+= "\nThe initial problem is\n";

        printTable();
        dummyCheck();
        printTable();
        steppingStoneMethod();*/

        writeToFile();
        JOptionPane.showMessageDialog(null, "Problem solved!");

        String filePath = "solution.txt";
        try {
            File file = new File(filePath);
            if (!Desktop.isDesktopSupported()) {
                System.out.println("Desktop not supported");
                return;
            }

            Desktop desktop = Desktop.getDesktop();
            if (file.exists()) {
                desktop.open(file);
            } else {
                System.out.println("File doesn't exist");
            }
        } catch (IOException e) {
            System.out.println("Error opening the file: " +
e.getMessage());
        }
        //testerPrinter();
    }

    public static void fillInitialArrays()
    {
        numberOfRows = supplyPoints + 1;
```

```
        numberOfColumns = demandPoints + 1;

        int listSize = textFields2.size();

        int i=0;
        int j=0;

        for (i=0; i<=40; i++)
        {
            if(i!=40)
            {
                supply[i]=0;
                demand[i]=0;
            }


            for (j=0; j<=40; j++)
            {
                if(j!=40 && i!=40)
                {
                    cost[i][j]=0;
                    solutionArray[i][j]=0;
                }
                problemArray[i][j]=0;

            }
        }

        i=0;
        j=0;
        for (int k=0;k<listSize;k++)
        {
            JTextField textFieldToGetValue = textFields2.get(k);

            if ((k+1)%numberOfColumns==0)
            {
                supply[i] =
Integer.parseInt(textFieldToGetValue.getText());
                problemArray[i][j] = supply[i];
                i++;
                j=0;
            }
            else if((k+1)>(numberOfRows-1)*numberOfColumns)
            {
                demand[j] =
Integer.parseInt(textFieldToGetValue.getText());
                problemArray[i][j] = demand[j];
                j++;
            }
            else
            {
                cost[i][j] =
Integer.parseInt(textFieldToGetValue.getText());
                problemArray[i][j] = cost[i][j];
                j++;
            }
```

```
    }
}

public static void printTable()
{
    allMessage +="\n";
    for (int i=0; i<numberOfRows;i++)
    {
        for (int j=0;j<numberOfColumns;j++)
        {
            if (i!=numberOfRows-1 || j!=numberOfColumns-1)
            {
                allMessage += problemArray[i][j];
                if (solutionArray[i][j] != 0)
                {
                    allMessage += "(" + solutionArray[i][j] + ")";
                }
                allMessage += "\t\t";
            }
        }
        allMessage +="\n";
    }
    allMessage +="\n";
}

public static void dummyCheck()
{

    int totalSupply, totalDemand;

    totalSupply = 0;
    totalDemand = 0;

    for (int i=0; i<supplyPoints;i++)
    {
        totalSupply += supply[i];
    }

    for (int i=0; i<demandPoints;i++)
    {
        totalDemand += demand[i];
    }

    if (totalSupply>totalDemand)
    {
        numberOfColumns++;
        demand[numberOfColumns-2]=totalSupply-totalDemand;
    }
    else if (totalSupply<totalDemand)
    {
        numberOfRows++;
        supply[numberOfRows-2]=totalDemand-totalSupply;
    }

    for (int i=0; i<numberOfRows;i++)
    {
```

```
            for (int j=0;j<numberOfColumns;j++)
            {
                if (i==numberOfRows-1 && j==numberOfColumns-1)
                {
                    continue;
                }
                else if (i==numberOfRows-1)
                {
                    problemArray[i][j]=demand[j];
                }
                else if (j==numberOfColumns-1)
                {
                    problemArray[i][j]=supply[i];
                }
                else
                {
                    problemArray[i][j]=cost[i][j];
                }
            }
        }

        if (totalSupply>totalDemand)
        {
            allMessage +="Total supply is " + totalSupply + " unit and
total demand is " + totalDemand + " unit\n";
            allMessage +="Thus, total supply > total demand. We add dummy
demand point!\n";
        }
        else if (totalSupply < totalDemand)
        {
            allMessage +="Total supply is " + totalSupply + " unit and
total demand is " + totalDemand + " unit\n";
            allMessage +="Thus, total supply < total demand. We add dummy
supply point!\n";
        }
        else
        {
            allMessage +="Total supply is " + totalSupply + " unit and
total demand is " + totalDemand + " unit\n";
            allMessage +="Thus, total supply = total demand. No need for
dummy point.\n";
        }
    }

    public static void northWestCornerMethod()
    {
        int i=0;
        int j=0;

        while (i<(numberOfRows-1) && j<(numberOfColumns-1))
        {

            allMessage +="Supply point " + (i+1) + " has " + supply[i] +
" unit free capacity and demand point " + (j+1) + " has " + demand[j] + "
unit unsatisfied demand\n";
            int quantity = min(supply[i], demand[j]);
```

```
            allMessage += "Minimum is " + quantity + "\n";

            solutionArray[i][j] = quantity;
            problemArray[i][(numberOfColumns-1)] -= quantity;
            problemArray[(numberOfRows-1)][j] -= quantity;
            supply[i] -= quantity;
            demand[j] -= quantity;

            if (supply[i]==0 && demand[j]==0)
            {
                allMessage += "This exhausts the capacity of supply point
" + (i+1) + " and meets the complete demand of demand point " + (j+1) +
"\n";
                printTable();
                i++;
                j++;
                if (i<(numberOfRows-1) && j<(numberOfColumns-1))
                    allMessage +="We go diagonal!\n";
            }
            else if (supply[i]==0)
            {
                allMessage += "This exhausts the capacity of supply point
" + (i+1) + " and leaves " + (demand[j]) + " unit demand of demand point
" + (j+1) + "\n";
                printTable();
                i++;
                if (i<(numberOfRows-1) && j<(numberOfColumns-1))
                    allMessage +="We go vertically!\n";
            }
            else
            {
                allMessage += "This meets the complete demand of demand
point " + (j+1) + " and leaves " + (supply[i]) + " unit supply of supply
point " + (i+1) + "\n";
                printTable();
                j++;
                if (i<(numberOfRows-1) && j<(numberOfColumns-1))
                    allMessage +="We go horizontally!\n";
            }

        }

        allMessage +="Solution with Northwest Corner Method is ";
        sumProduct();
    }

    public static void sumProduct()
    {

        int totalCost = 0;

        for (int i=0; i<numberOfRows-1;i++)
        {
            for (int j=0; j<numberOfColumns-1;j++)
            {
                totalCost += problemArray[i][j] * solutionArray[i][j];
```

```
            }
        }

        allMessage += totalCost + "\n";
    }

    public static void writeToFile()
    {
        try
        {
            FileWriter myWriter = new FileWriter("solution.txt");
            myWriter.write(allMessage);
            myWriter.close();
        }
        catch (IOException e)
        {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }

    public static void leastCostMethod()
    {

        int minI, minJ;
        minI = -2;
        minJ = -2;
        while(minI != -1 && minJ != -1)
        {
            int minCost = -1;
            int maxAllocation = 0;
            minI = -1;
            minJ = -1;

            for (int i=0;i<(numberOfRows-1);i++)
            {
                if (supply[i]==0)
                    continue;

                for (int j=0;j<(numberOfColumns-1);j++)
                {
                    if (demand[j]==0)
                        continue;

                    if (solutionArray[i][j]==0 && minCost==-1)
                    {
                        minCost = cost[i][j];
                        minI = i;
                        minJ = j;
                        maxAllocation = min(supply[i], demand[j]);
                    }
                    else if (solutionArray[i][j]==0 && cost[i][j] <
minCost)
                    {
                        minCost = cost[i][j];
                        minI = i;
```

```
                        minJ = j;
                        maxAllocation = min(supply[i], demand[j]);
                    }
                    else if(solutionArray[i][j]==0 && cost[i][j]==minCost
&& maxAllocation < min(supply[i], demand[j]))
                    {
                        minCost = cost[i][j];
                        minI = i;
                        minJ = j;
                        maxAllocation = min(supply[i], demand[j]);
                    }
                }
            }

            if (minI == -1 || minJ == -1)
            {
                break;
            }

            int quantity = min(supply[minI], demand[minJ]);
            allMessage += "Min cost is " + minCost + " at supply point "
+ (minI+1) + " and demand point " + (minJ+1) + " with maximum allocation
amount " + quantity + "\n";
            solutionArray[minI][minJ] = quantity;
            supply[minI] -= quantity;
            demand[minJ] -= quantity;
            problemArray[minI][numberOfColumns-1] -= quantity;
            problemArray[numberOfRows-1][minJ] -= quantity;
            if (supply[minI]==0 && demand[minJ]==0)
            {
                allMessage += "This exhausts the capacity of supply point
" + (minI+1) + " and meets the complete demand of demand point " +
(minJ+1) + "\n";
            }
            else if (supply[minI]==0)
            {
                allMessage += "This exhausts the capacity of supply point
" + (minI+1) + " and leaves " + (demand[minJ]) + " unit demand of demand
point " + (minJ+1) + "\n";
            }
            else
            {
                allMessage += "This meets the complete demand of demand
point " + (minJ+1) + " and leaves " + (supply[minI]) + " unit supply of
supply point " + (minI+1) + "\n";
            }
            printTable();
        }

        allMessage +="Solution with Least Cost Method is ";
        sumProduct();
    }

    public static void vogelsApproximationMethod()
    {
```

```
int myCheck=1;
int rowMin1, rowMin2, colMin1, colMin2;
int maxPenalty, maxAllocation, minCost, selectedI, selectedJ;
int maxAllocation2, minCost2, selectedI2, selectedJ2;

while (myCheck!=0)
{
    maxPenalty =-1;
    maxAllocation = -1;
    minCost=-1;
    selectedI=-1;
    selectedJ=-1;

    for (int i=0;i<numberOfRows-1;i++)
        rowPenalty[i]=-1;

    for (int i=0; i<numberOfColumns-1;i++)
        colPenalty[i]=-1;

    for (int i=0; i<numberOfRows-1;i++)
    {
        if (supply[i]==0)
            continue;

        rowMin1=-1;
        rowMin2=-1;

        for (int j=0; j<numberOfColumns-1;j++)
        {
            if (demand[j]==0)
                continue;

            if (rowMin1==-1)
            {
                rowMin1=cost[i][j];
            }
            else if (cost[i][j]<=rowMin1)
            {
                rowMin2=rowMin1;
                rowMin1=cost[i][j];
            }
            else if(rowMin2==-1)
            {
                rowMin2=cost[i][j];
            }
            else if (cost[i][j]<rowMin2)
            {
                rowMin2=cost[i][j];
            }
        }

        if (rowMin2==-1)
        {
            rowPenalty[i]=rowMin1;
        }
        else
```

```
        {
            rowPenalty[i]=rowMin2-rowMin1;
        }
    }

    for (int j=0; j<numberOfColumns-1;j++)
    {
        if (demand[j]==0)
            continue;

        colMin1=-1;
        colMin2=-1;

        for (int i=0; i<numberOfRows-1;i++)
        {
            if (supply[i]==0)
                continue;

            if (colMin1==-1)
            {
                colMin1=cost[i][j];
            }
            else if (cost[i][j]<=colMin1)
            {
                colMin2=colMin1;
                colMin1=cost[i][j];
            }
            else if(colMin2==-1)
            {
                colMin2=cost[i][j];
            }
            else if (cost[i][j]<colMin2)
            {
                colMin2=cost[i][j];
            }
        }

        if (colMin2==-1)
        {
            colPenalty[j]=colMin1;
        }
        else
        {
            colPenalty[j]=colMin2-colMin1;
        }
    }

    for (int i=0; i<numberOfRows-1;i++)
    {
        if (supply[i]==0)
            continue;

        if(maxPenalty==-1 || rowPenalty[i]>maxPenalty)
        {
            for (int j=0;j<numberOfColumns-1;j++)
            {
```

```
                        if (demand[j] == 0)
                            continue;

                        if ((minCost==-1) || (rowPenalty[i]>maxPenalty)
|| (cost[i][j]<minCost) || (minCost==cost[i][j] &&
maxAllocation<min(supply[i], demand[j])))
                        {
                            maxPenalty = rowPenalty[i];
                            maxAllocation = min(supply[i], demand[j]);
                            minCost = cost[i][j];
                            selectedI=i;
                            selectedJ=j;
                        }
                    }

                }
                else if (maxPenalty == rowPenalty[i])
                {
                    minCost2=-1;
                    maxAllocation2=-1;
                    selectedI2=-1;
                    selectedJ2=-1;
                    for (int j=0;j<numberOfColumns-1;j++)
                    {
                        if (demand[j] == 0)
                            continue;
                        if ((minCost2==-1) || (cost[i][j]<minCost2) ||
(minCost2==cost[i][j] && maxAllocation2<min(supply[i], demand[j])))
                        {
                            maxAllocation2 = min(supply[i], demand[j]);
                            minCost2 = cost[i][j];
                            selectedI2=i;
                            selectedJ2=j;
                        }

                    }

                    if (maxAllocation2>maxAllocation)
                    {
                        maxAllocation = maxAllocation2;
                        minCost = minCost2;
                        selectedI = selectedI2;
                        selectedJ = selectedJ2;
                    }

                }
            }

            for (int j=0; j<numberOfColumns-1;j++)
            {
                if (demand[j]==0)
                    continue;

                if(colPenalty[j]>maxPenalty)
                {
                    for (int i=0;i<numberOfRows-1;i++)
```

```
                    {
                        if (supply[i] == 0)
                            continue;

                        if ((colPenalty[j]>maxPenalty) ||
(cost[i][j]<minCost) || (minCost==cost[i][j] &&
maxAllocation<min(supply[i], demand[j])))
                        {
                            maxPenalty = colPenalty[j];
                            maxAllocation = min(supply[i], demand[j]);
                            minCost = cost[i][j];
                            selectedI=i;
                            selectedJ=j;
                        }
                    }

                }
                else if (maxPenalty == colPenalty[j])
                {
                    minCost2=-1;
                    maxAllocation2=-1;
                    selectedI2=-1;
                    selectedJ2=-1;
                    for (int i=0;i<numberOfRows-1;i++)
                    {
                        if (supply[i] == 0)
                            continue;

                        if ((minCost2==-1) || (cost[i][j]<minCost2) ||
(minCost2==cost[i][j] && maxAllocation2<min(supply[i], demand[j])))
                        {
                            maxAllocation2 = min(supply[i], demand[j]);
                            minCost2 = cost[i][j];
                            selectedI2=i;
                            selectedJ2=j;
                        }

                    }

                    if (maxAllocation2>maxAllocation)
                    {
                        maxAllocation = maxAllocation2;
                        minCost = minCost2;
                        selectedI = selectedI2;
                        selectedJ = selectedJ2;
                    }
                }
            }

            int quantity = min(supply[selectedI], demand[selectedJ]);
            allMessage+="Max penalty is " + maxPenalty;

            if (rowPenalty[selectedI]==maxPenalty)
                allMessage+=" in row " + (selectedI+1) + "\n";
            else
                allMessage+=" in column " + (selectedJ+1) + "\n";
```

```
            allMessage += "Min cost is " + minCost + " at supply point "
+ (selectedI+1) + " and demand point " + (selectedJ+1) + " with maximum
allocation amount " + quantity + "\n";
            solutionArray[selectedI][selectedJ] = quantity;
            supply[selectedI] -= quantity;
            demand[selectedJ] -= quantity;
            problemArray[selectedI][numberOfColumns-1] -= quantity;
            problemArray[numberOfRows-1][selectedJ] -= quantity;

            if (supply[selectedI]==0 && demand[selectedJ]==0)
            {
                allMessage += "This exhausts the capacity of supply point
" + (selectedI+1) + " and meets the complete demand of demand point " +
(selectedJ+1) + "\n";
            }
            else if (supply[selectedI]==0)
            {
                allMessage += "This exhausts the capacity of supply point
" + (selectedI+1) + " and leaves " + (demand[selectedJ]) + " unit demand
of demand point " + (selectedJ+1) + "\n";
            }
            else
            {
                allMessage += "This meets the complete demand of demand
point " + (selectedJ+1) + " and leaves " + (supply[selectedI]) + " unit
supply of supply point " + (selectedI+1) + "\n";
            }
            printTable2();

            myCheck = 0;
            for (int i=0;i<numberOfRows-1;i++)
            {
                if (supply[i]>0)
                {
                    myCheck=1;
                    break;
                }
            }
        }
        allMessage +="Solution with Vogel's Approximation Method is ";
        sumProduct();

    }

    public static void printTable2()
    {
        allMessage +="\n";
        for (int i=0; i<numberOfRows;i++)
        {
            for (int j=0;j<numberOfColumns;j++)
            {
                if (i!=numberOfRows-1 || j!=numberOfColumns-1)
                {
                    allMessage += problemArray[i][j];
                    if (solutionArray[i][j] != 0)
```

```
                {
                    allMessage += "(" + solutionArray[i][j] + ")";
                }
                allMessage += "\t\t";
            }
        }
        if (i<numberOfRows-1)
            allMessage +=rowPenalty[i]+ "\n";
        else
            allMessage +="\n";
    }

    for (int j=0;j<numberOfColumns-1;j++)
    {
        allMessage += colPenalty[j];
        allMessage += "\t\t";
    }
    allMessage +="\n\n";
}

public static void russellsApproximationMethod()
{
    int myCheck=0;
    int minCost;
    int selectedI, selectedJ;
    selectedI=0;
    selectedJ=0;

    while(myCheck!=-1)
    {

        for (int i=0;i<numberOfRows-1;i++)
            rowPenalty[i]=-1;

        for (int i=0; i<numberOfColumns-1;i++)
            colPenalty[i]=-1;

        for (int i=0;i<numberOfRows-1;i++)
        {
            if (supply[i]==0)
                continue;

            for (int j=0; j<numberOfColumns-1;j++)
            {
                if (demand[j]==0)
                    continue;

                if (rowPenalty[i]<cost[i][j])
                    rowPenalty[i] = cost[i][j];

            }
        }

        for (int j=0;j<numberOfColumns-1;j++)
        {
            if (demand[j]==0)
```

```
                    continue;

                for (int i=0; i<numberOfRows-1;i++)
                {
                    if (supply[i]==0)
                        continue;

                    if (colPenalty[j]<cost[i][j])
                        colPenalty[j] = cost[i][j];
                }
            }

            minCost = 1;

            for (int i=0;i<numberOfRows-1;i++)
            {
                if (supply[i]==0)
                    continue;

                for (int j=0; j<numberOfColumns-1;j++)
                {
                    if (demand[j]==0)
                        continue;

                    allMessage+="Delta(" + (i+1) + "," + (j+1) + ")=c(" +
(i+1) + "," + (j+1) + ")-(U(" + (i+1) + ")+V(" + (j+1) + ")=" +
cost[i][j] + "-(" + rowPenalty[i] + "+" + colPenalty[j] + ")=" +
(cost[i][j]-(rowPenalty[i] + colPenalty[j])) + "\n";

                    if (minCost>(cost[i][j]-(rowPenalty[i] +
colPenalty[j])))
                    {
                        minCost = (cost[i][j]-(rowPenalty[i] +
colPenalty[j]));
                        selectedI = i;
                        selectedJ = j;
                    }
                }
            }

            int quantity = min(supply[selectedI], demand[selectedJ]);
            allMessage+="Min delta is " + minCost + " at supply point " +
(selectedI+1) + " and demand point " + (selectedJ+1) + " with maximum
allocation amount " + quantity + "\n";

            solutionArray[selectedI][selectedJ] = quantity;
            supply[selectedI] -= quantity;
            demand[selectedJ] -= quantity;
            problemArray[selectedI][numberOfColumns-1] -= quantity;
            problemArray[numberOfRows-1][selectedJ] -= quantity;

            if (supply[selectedI]==0 && demand[selectedJ]==0)
            {
                allMessage += "This exhausts the capacity of supply point
" + (selectedI+1) + " and meets the complete demand of demand point " +
(selectedJ+1) + "\n";
```

```
            }
            else if (supply[selectedI]==0)
            {
                allMessage += "This exhausts the capacity of supply point
" + (selectedI+1) + " and leaves " + (demand[selectedJ]) + " unit demand
of demand point " + (selectedJ+1) + "\n";
            }
            else
            {
                allMessage += "This meets the complete demand of demand
point " + (selectedJ+1) + " and leaves " + (supply[selectedI]) + " unit
supply of supply point " + (selectedI+1) + "\n";
            }
            printTable2();
            myCheck=-1;

            for (int i=0;i<numberOfRows-1;i++)
            {
                if (supply[i]>0)
                {
                    myCheck=0;
                    break;
                }
            }
        }

        allMessage +="Solution with Russell's Approximation Method is ";
        sumProduct();
    }

    public static void modiMethod()
    {
        int myControl=1;
        int myControl2=1;
        int rowOrCol=-1;
        int maxNumAllocation=0;
        int currentAllocation=0;
        int relatedRowOrCol=-1;
        int i1,j1,i2,j2,i3,j3,i4,j4;
        int mindij;
        int minAllocation=0;
        i1=-1;
        j1=-1;
        i2=-1;
        j2=-1;
        i3=-1;
        j3=-1;
        i4=-1;
        j4=-1;

        allMessage+="First, we find an initial basic feasible solution by
using Vogel's Approximation Method \n";
        vogelsApproximationMethod();
        allMessage+="\n";

        while (myControl==1)
```

```
        {
            maxNumAllocation=0;

            for (int i=0;i<numberOfRows-1;i++)
            {
                currentAllocation=0;

                for (int j=0; j<numberOfColumns-1;j++)
                {
                    if (solutionArray[i][j]>0)
                    {
                        currentAllocation++;
                    }
                }

                if (currentAllocation>maxNumAllocation)
                {
                    maxNumAllocation = currentAllocation;
                    rowOrCol = 1;
                    relatedRowOrCol=i;
                }
            }

            for (int j=0; j<numberOfColumns-1;j++)
            {
                currentAllocation=0;

                for (int i=0;i<numberOfRows-1;i++)
                {
                    if (solutionArray[i][j]>0)
                    {
                        currentAllocation++;
                    }
                }

                if (currentAllocation>maxNumAllocation)
                {
                    maxNumAllocation = currentAllocation;
                    rowOrCol = 2;
                    relatedRowOrCol=j;
                }
            }

            allMessage+="Maximum number of allocation is ";
            if(rowOrCol==1)
                allMessage+="in row ";
            else
                allMessage+="in column";
            allMessage+=" " + (relatedRowOrCol+1) + " thus, substituting
";
            if(rowOrCol==1)
                allMessage+="u(" + (relatedRowOrCol+1) + ")=0, we
get:\n";
            else
                allMessage+="v(" + (relatedRowOrCol+1) + ")=0, we
get:\n";
```

```
            for (int i=0;i<numberOfRows-1;i++)
                supply[i]=0;

            for(int j=0;j<numberOfColumns-1;j++)
                demand[j]=0;

            if (rowOrCol==1)
            {
                rowPenalty[relatedRowOrCol]=0;
                supply[relatedRowOrCol]=1;
            }
            else
            {
                colPenalty[relatedRowOrCol]=0;
                demand[relatedRowOrCol]=1;
            }

            myControl2=1;
            while (myControl2==1)
            {
                for (int i=0;i<numberOfRows-1;i++)
                {
                    if (supply[i]==0)
                        continue;

                    for (int j=0; j<numberOfColumns-1;j++)
                    {
                        if (demand[j]==1)
                            continue;

                        if(solutionArray[i][j]>0 && demand[j]==0)
                        {
                            colPenalty[j]=cost[i][j]-rowPenalty[i];
                            demand[j]=1;
                            allMessage +="v(" + (j+1) + ") = c(" + (i+1)
+ "," + (j+1) + ") - u(" + (i+1) + ")--> " + cost[i][j] + " - " +
rowPenalty[i] + " = " + colPenalty[j] + "\n";
                        }
                    }
                }

                for (int j=0;j<numberOfColumns-1;j++)
                {
                    if (demand[j]==0)
                        continue;

                    for (int i=0; i<numberOfRows-1;i++)
                    {
                        if (supply[i]==1)
                            continue;

                        if(solutionArray[i][j]>0)
                        {
                            rowPenalty[i]=cost[i][j]-colPenalty[j];
                            supply[i]=1;
```

```
                                allMessage +="u(" + (i+1) + ") = c(" + (i+1)
+ "," + (j+1) + ") - v(" + (j+1) + ")--> " + cost[i][j] + " - " +
colPenalty[j] + " = " + rowPenalty[i] + "\n";
                            }
                        }
                    }

                    myControl2=0;
                    for (int i=0; i<numberOfRows-1;i++)
                    {
                        if(supply[i] == 0)
                        {
                            myControl2=1;
                            break;
                        }
                    }

                    if (myControl2==0)
                    {
                        for (int j=0;j<numberOfColumns-1;j++)
                        {
                            if(demand[j]==0)
                            {
                                myControl2=1;
                                break;
                            }
                        }
                    }
                }

                printTable2();

                myControl=0;
                mindij=0;
                for (int i=0;i<numberOfRows-1;i++)
                {
                    for (int j=0;j<numberOfColumns-1;j++)
                    {
                        if (solutionArray[i][j]==0)
                        {
                            allMessage+="d(" + (i+1) + "," + (j+1) + ") = c("
+ (i+1) + "," + (j+1) + ") - (u(" + (i+1) + ") + v(" + (j+1) + ")) = " +
cost[i][j] + " - (" + rowPenalty[i] + " + " + colPenalty[j] + ") = " +
(cost[i][j] - rowPenalty[i] - colPenalty[j]) + "\n";
                        }

                        if (cost[i][j] - rowPenalty[i] - colPenalty[j]<0)
                        {
                            myControl=1;
                            if (mindij>cost[i][j] - rowPenalty[i] -
colPenalty[j])
                            {
                                mindij = cost[i][j] - rowPenalty[i] -
colPenalty[j];
                                i1=i;
                                j1=j;
```

```
                        }
                    }
                }
            }

            if (myControl==0)
            {
                allMessage+="All d(i,j) values are non-negative. The
solution is optimal.\n";
                printTable();
                continue;
            }

            allMessage+="The minimum negative value from all d(i,j)
(opportunity cost) = d(" + (i1+1) + "," + (j1+1) + ") = " + mindij +
"\n";

            for (int i=0; i<numberOfRows-1;i++)
            {
                if(solutionArray[i][j1]!=0)
                {
                    for(int j=0; j<numberOfColumns-1;j++)
                    {
                        if(solutionArray[i1][j]!=0 &&
solutionArray[i][j]!=0)
                        {
                            i2=i1;
                            i3=i;
                            i4=i;
                            j2=j;
                            j3=j1;
                            j4=j;
                            minAllocation = min(solutionArray[i2][j2],
solutionArray[i3][j3]);
                        }
                    }
                }
            }

            allMessage+="The closed loop is S" + (i1+1) + "D" + (j1+1) +
" - S" + (i2+1) + "D" + (j2+1) + " - S" + (i4+1) + "D" + (j4+1) + " - S"
+ (i3+1) + "D" + (j3+1) + "\n";
            allMessage+="We can allocate " + minAllocation + " unit from
S" + (i2+1) + "D" + (j2+1) + " and S" + (i3+1) + "D" + (j3+1) + " to S" +
+ (i1+1) + "D" + (j1+1) + " and S" + (i4+1) + "D" + (j4+1) + "\n";
            solutionArray[i1][j1] += minAllocation;
            solutionArray[i2][j2] -= minAllocation;
            solutionArray[i3][j3] -= minAllocation;
            solutionArray[i4][j4] += minAllocation;
            allMessage+="New solution is: \n";
            printTable();
            allMessage+="We continue with the next iteration!\n";
        }

        allMessage +="Optimal solution with MODI Method is ";
        sumProduct();
```

```
    }

    public static void steppingStoneMethod()
    {
        int myControl=1;
        int minCost=0;
        int maxAllocation = 0;
        int totalCost = 0;

        int i1=-1;
        int i2=-1;
        int i3=-1;
        int i4=-1;

        int j1=-1;
        int j2=-1;
        int j3=-1;
        int j4=-1;

        int fi1=-1;
        int fi2=-1;
        int fi3=-1;
        int fi4=-1;

        int fj1=-1;
        int fj2=-1;
        int fj3=-1;
        int fj4=-1;

        allMessage+="First, we find an initial basic feasible solution by
using Vogel's Approximation Method \n";
        vogelsApproximationMethod();
        allMessage+="\n";

        while (myControl==1)
        {
            myControl = 0;
            minCost = 0;
            allMessage += "Create closed loop for unoccupied cells, we
get:\n";

            for (int i=0;i<numberOfRows-1;i++)
            {
                for (int j=0; j<numberOfColumns-1;j++)
                {
                    if (solutionArray[i][j]==0)
                    {
                        i1=i;
                        j1=j;

                        for (int ii=0; ii<numberOfRows-1;ii++)
                        {
                            if(solutionArray[ii][j1]!=0)
                            {
                                for(int jj=0; jj<numberOfColumns-1;jj++)
                                {
```

```
                                    if(solutionArray[i1][jj]!=0 &&
solutionArray[ii][jj]!=0)
                                    {
                                        i2=i1;
                                        i3=ii;
                                        i4=ii;

                                        j2=jj;
                                        j3=j1;
                                        j4=jj;

                                        totalCost = cost[i1][j1] -
cost[i2][j2] - cost[i3][j3] + cost[i4][j4];

                                        allMessage+="S" + (i1+1) + "D" +
(j1+1) + " - S" + (i2+1) + "D" + (j2+1) + " - S" + (i4+1) + "D" + (j4+1)
+ " - S" + (i3+1) + "D" + (j3+1) + " with total cost " + totalCost +
"\n";

                                        if (totalCost<minCost)
                                        {
                                            myControl=1;
                                            fi1=i1;
                                            fi2=i2;
                                            fi3=i3;
                                            fi4=i4;
                                            fj1=j1;
                                            fj2=j2;
                                            fj3=j3;
                                            fj4=j4;
                                            minCost = totalCost;
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }

            if (myControl==0)
            {
                allMessage+="All cost values are non-negative. The
solution is optimal.\n";
                printTable();
                continue;
            }

            maxAllocation = min(solutionArray[fi2][fj2],
solutionArray[fi3][fj3]);

            allMessage+="We have negative cost, thus the solution is not
optimal.\n";
            allMessage+="The min cost closed loop is S" + (fi1+1) + "D" +
(fj1+1) + " - S" + (fi2+1) + "D" + (fj2+1) + " - S" + (fi4+1) + "D" +
(fj4+1) + " - S" + (fi3+1) + "D" + (fj3+1) + "\n";
```

```
            allMessage+="We can allocate " + maxAllocation + " unit from
S" + (fi2+1) + "D" + (fj2+1) + " and S" + (fi3+1) + "D" + (fj3+1) + " to
S" + + (fi1+1) + "D" + (fj1+1) + " and S" + (fi4+1) + "D" + (fj4+1) +
"\n";
            solutionArray[fi1][fj1] += maxAllocation;
            solutionArray[fi2][fj2] -= maxAllocation;
            solutionArray[fi3][fj3] -= maxAllocation;
            solutionArray[fi4][fj4] += maxAllocation;
            allMessage+="New solution is: \n";
            printTable();
            allMessage+="We continue with the next iteration!\n";

        }
        allMessage +="Optimal solution with Stepping Stone Method is ";
        sumProduct();
    }



    public static void testerPrinter()
    {
        String s1="";
        String s2="";
        String s3="";
        String s4="";
        String s5="";
        String s6="";
        String s7="";
        String s8="";
        String s9="";

        s1 = "Number of supply points is " + supplyPoints;
        s2 = "Number of demand points is " + demandPoints;
        s3 = "Number of rows " + numberOfRows;
        s4 = "Number of columns " + numberOfColumns;

        s5 = "Supply array [";
        for (int i=0; i<numberOfRows-1;i++)
        {
            s5 += supply[i] + " ";
        }
        s5 += "]";

        s6 = "Demand array [";
        for (int i=0; i<numberOfColumns-1;i++)
        {
            s6 += demand[i] + " ";
        }
        s6 += "]";

        s7 = "Cost array [";
        for (int i=0; i<numberOfRows-1;i++)
        {
            for (int j=0; j<numberOfColumns-1;j++)
            {
                s7 += cost[i][j] + " ";
```

```
            }
            s7 += "\n";
        }
        s7 += "]";

        s8 = "Problem array [";
        s9 = "Solution array [";
        for (int i=0; i<numberOfRows;i++)
        {
            for (int j=0; j<numberOfColumns;j++)
            {
                s8 += problemArray[i][j] + " ";
                s9 += solutionArray[i][j] + " ";
            }
            s8 += "\n";
            s9 += "\n";
        }
        s8 += "]";
        s9 += "]";

        JOptionPane.showMessageDialog(null, s1);
        JOptionPane.showMessageDialog(null, s2);
        JOptionPane.showMessageDialog(null, s3);
        JOptionPane.showMessageDialog(null, s4);
        JOptionPane.showMessageDialog(null, s5);
        JOptionPane.showMessageDialog(null, s6);
        JOptionPane.showMessageDialog(null, s7);
        JOptionPane.showMessageDialog(null, s8);
        JOptionPane.showMessageDialog(null, s9);
    }
}
```