Development of an Interactive Software System for Solving the Transportation Problem

By: Adam Wermes
Advisor: Dr. Betty Hickman

March 30, 2007
University of Nebraska at Omaha

Kerrigan Research Minigrant Program
## Table of Contents

Abstract .................................................. 1
Farmer Jones example .................................. 2
Transportation Problem Description .............. 3
Program Language ........................................ 6
The Input Form ........................................... 6
The Bipartite Graph ..................................... 10
The Transportation Tableau ......................... 13
Transportation Tableau Cost Edit .................... 14
Balancing the Transportation Problem ............. 15
The Northwest Corner Rule ......................... 16
The Minimum Cost Method ......................... 17
Vogel’s Method ........................................... 18
Optimality Check ....................................... 19
The Transportation Simplex ......................... 21
References .............................................. 26
Appendix A, Flowcharts
Appendix B, Selected Functions
Abstract

The transportation problem is a specialized problem in Operations Research. The purpose of this paper is to discuss the development of a program designed to solve the transportation problem. The program is used as an educational tool to learn how to solve the transportation problem and can be used as a tool to solve the problem without the educational parts. The logical development of the program includes a discussion of the steps to solve the transportation problem. The first step in the process is the collection of the information about the transportation problem. After collecting the data check to see if the problem is balanced, look for a basic feasible solution, then check to see if it is optimal. A detailed discussion of each step is included, but the focus is on the development of the program. The code and program flow is the central theme of this paper. Screen shots from the program are included as well as code and flow to understand the processes of the program and how it relates to the solution of the transportation problem.
In the field of Operations Research, problems are described by a series of linear inequalities. A single inequality is called a constraint. It includes variables that are related to each other and describes limitations placed on them. The collection of constraints and variables is called a linear programming problem (LP).

**Farmer Jones Example**

As a brief example of an LP, consider the “Farmer Jones” problem in *Introduction to Mathematical Programming* by Winston and Venkataramanan. Farmer Jones wants to know how many acres of corn and wheat to plant. Each acre of land can produce 25 bushels of wheat or 10 bushels of corn. Each acre in wheat requires 10 hours of labor per week. Alternatively, an acre in corn requires 4 hours of labor per week. Wheat is sold at 4 dollars per bushel and corn is sold at 3 dollars per bushel. There are 7 acres of land and 40 hours of labor per week available to farmer Jones. The government says that he must produce at least 30 bushels of corn this year.

To attack this problem, we first define our variables. We might let $x_1$ equal acres of corn and $x_2$ equal acres of wheat.

Next we define the constraints. In this problem there are 7 acres of land available. This gives us our first constraint, on land, $x_1 + x_2 \leq 7$. Farmer Jones can work up to 40 hours per week. Given the time requirements for each crop, we see that the labor constraint is $4x_1 + 10x_2 \leq 40$. The government requires that Jones produce at least 30 bushels of corn this year, or $10x_1 \geq 30$. Our final constraints are that both $x_1$ and $x_2$ must be positive numbers, or $x_1 \geq 0$ and $x_2 \geq 0$.

To determine optimal production levels for each crop we define an objective function. Farmer Jones wants to maximize profit, $Z$, where $Z = 30x_1 + 100x_2$. This LP has a large number of possible solutions of the form $(x_1, x_2)$ within its solution region, but only four of these need be considered. Substituting them in turn into the objective function gives us the optimal solution. For Farmer Jones, an example of a suboptimal solution is $(5, 2)$. This gives a value to the objective function of 350. The optimal solution $(3, 2.8)$. This gives an objective function value of 370.
The Farmer Jones problem is typical of a simple LP. More realistic LP’s may have hundreds of constraints and variables.

Depending on the size of the linear programming problem, there can be more than one way to solve it. LP’s with only two or three variables may be solved graphically. Obviously, it is impossible to graph constraints with more than three variables, so these LP’s get their solutions from an algorithm called the simplex algorithm. The simplex algorithm works on LP’s with many constraints and many variables. There could be hundreds or even thousands of each. As an LP gets larger, so does the time to implement the simplex algorithm. So, in the spirit of finding an easier way to do things, several other methods have been found as alternatives to the simplex algorithm.

Transportation Problem Description

There is one particular problem in Operations Research that has been repeatedly studied because of its numerous real world applications. It is the transportation problem. The transportation problem is an LP, and people have found quick and creative ways to solve it.

What are the differences between a typical LP and the transportation problem? First, consider what the transportation problem represents in the real world. As its name suggests, the transportation problem deals with transporting some product, tangible or intangible, from a supply point to a demand point. The problem lies in figuring out how much of each product should go to each demand point from each supply point and finding the best way to move the products. (There are usually several ways to move products from supply points to demand points.)

Just as in the Farmer Jones example, there can be more than one solution to the transportation problem, but which is optimal? That is, which solution is the cheapest? To determine the best, cheapest way to move products, one takes into account the two most important pieces in the transportation problem, the cost associated between each supply and demand point and the quantities associated with each supply and demand point. The cost that exists for each connection between each supply point and each demand point represents a cost in moving an amount of product from the supply point to a particular demand point.
Collectively, the costs are placed into a cost matrix. The combination of the list of supply points, demand points, and the cost matrix make up the specialized transportation tableau. The transportation tableau is central to finding the solution of the transportation problem by way of this specialized method. Before introducing the program and how to solve the transportation problem with the transportation tableau, first consider the transportation problem definition in terms of variables and constraints and then some of the properties of the transportation problem.

When building the LP for the transportation problem, picture the cost matrix. Each place in the cost matrix represents a cost associated with moving a unit of goods from a supply point to a demand point. There is also a variable associated with each element in the cost matrix that represents how many units should be moved for that particular supply point to a particular demand point. A 2 x 3 matrix has six variables, two rows and three columns; similarly a 3 x 4 matrix has twelve variables, three rows and four columns of variables. The first set of constraints is called supply constraints. Supply constraints are obtained by summing each row and setting that sum to be less than or equal to the supply quantity associated with each row. Similarly, the sum of each column will be set to be greater than or equal to the demand quantity associated with each column. Finally, there is the objective function that is determined by multiplying each cost in the cost matrix by each associated variable and taking the sum. That sum is then set it equal to some variable, usually labeled $Z$.

With an understanding of the shorthand notation for the transportation LP, figuring out the long version is not too difficult. The short version has a coefficient in front of each variable in the supply and demand constraints that is listed in the cost matrix. The long version of the supply constraints consists of those same variables and coefficients and includes all other variable from the other supply constraints, but with a zero coefficient. The same is true for the demand constraints.

The LP that represents the transportation problem can now be run through the simplex algorithm. Consider that a four by three matrix has seven constraints with 12 variables each. Because of the usually large number of variables and constraints, the simplex method is too time consuming for solving transportation problems. A specialized solution is required.
The real world operation modeled by the transportation LP is the movement of products from one point to another. In the cost matrix the cost of moving each good is listed, and the objective is to minimize the total cost of meeting the needs of the demand points while not exceeding quantities available at the supply points. Any representation of a quantity and an associated cost to move it can be represented by the transportation problem. Consider the “Powerco” example taken from Introduction to Mathematical Programming by Winston and Venkataramanan. The following matrix is the cost matrix associated with the Powerco company.

\[
\begin{bmatrix}
8 & 6 & 10 & 9 \\
9 & 12 & 13 & 7 \\
14 & 9 & 16 & 5 \\
\end{bmatrix}
\]

The next two matrices are the supply quantity matrix, on the left, and the demand quantity matrix, on the right.

\[
\begin{bmatrix}
35 \\
50 \\
40 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
45 & 20 & 30 & 30 \\
\end{bmatrix}
\]

The supply points are labeled plants 1 through 3 and the demand points are labeled cities 1 through 4. To get one unit of electricity from plant 1 to city 1 costs 8 dollars, 12 from plant 2 to city 2, and so on. City 1 needs 35 units of electricity while plant 1 can provide 45 units of electricity. The problem is “balanced”, because the sum of the supply quantities equals the sum of the demand quantities.

I am concerned here less with a discussion of transportation problems like this than with the development of a program that is used to solve them. Keep in mind that my program does not just churn out answers to the transportation problem. There are several commercial software packages available to
do just that. This program is a way to learn how to solve the transportation problem and practice developing solutions to it. (Even though the program is intended for education and troubleshooting transportation problem solutions, there is an option to jump to the solution for a given problem without having to click through all the methods.)

**Program Language Description**

Development of the program was done in an environment called Microsoft Visual Basic 6 (herein referred to as VB6). Due to the simple nature of the drawings associated with the transportation problem, VB6 was adequate. Other choices to develop this program would have been Java or C++. In order to write the program in Java there would need to be an environment capable of drawing, so the Java windows methods would have to be used. To have written the program in C++ would again require graphics capable functions. One such package with said function is Microsoft Visual C++ 6.0, but there are others. Any C++ package that is intended to develop windows software would work. The C++ and Java environments would probably have made development longer and more difficult due to the more complex nature of each language, but each has its own advantages. C++ would run with less overhead and therefore execute the program faster, while Java would allow the program to execute on more computers easily. Most programming languages are inherently command line based and C++ and Java are no exception. To do graphics with these languages requires extensions of the language that can be tricky and time consuming to implement. A VB6 program starts with a blank window, so the complexity is bypassed completely and one can immediately begin writing code and drawing the necessary shapes. Because the program output is comprised mainly of squares and lines, it has no need for complicated graphics functions. The basic windows environment of VB6 serves up enough useful graphics functions to get the job done.

**The input Form**

When writing any program the best place to start is at the beginning. In order to create some output, first decide what the input will be, what it will look like, and how it will be retrieved. In the case
of the transportation problem there will be several inputs. Take a look at image one, it is the first form the user will see and it contains all the input areas that are about to be discussed.

Image 1

Image 1 is the empty form before it has been populated with data from the transportation problem. The first input will be the name of the company, which the solution of the transportation problem will represent. This will be a string of characters, so a textbox suffices. Similarly, a textbox will suffice for the names of the supply and demand points, and they will have the same type of input, a string of characters. When inputting the quantities for the supply and demand points, textboxes will be used again, but the input will be limited to positive integers. This will be the first chance to do some data checking. The quantities for supply and demand points represent quantities which can only be expressed as positive
whole numbers. It is important to confirm early on that the quantities are positive integer values because they will be used throughout the program and they need to be correct from the very beginning.

When defining a specific transportation problem, the input for the names and quantities of the supply and demand points is only the beginning. With the first set of inputs describing the transportation problem at hand, the data will be used to create input in other areas. More inputs will come in the form of the cost matrix and the quantity of each supply and demand point, which will also be matrices. The most obvious representation of these matrices is by grids. The grid that will be used repeatedly throughout the program is called MSFlexgrid. The MSFlexgrid is typically used for displaying information from the program, but there are three areas where it is needed for input. The grid will be the easiest way to get the values for the aforementioned matrices. This is unfortunate, because, despite VB6 having a few built in grids, none of them have natural features which make them directly editable. In order to use an MSFlexgrid for data entry there will be some unusual manipulations with a textbox. The textbox will be of the same type as those used to get the supply and demand quantities, but its properties will be changed so that it is invisible. The textbox will be the exact height and width of each individual grid cell, and it will have no border, so that the resulting cell will appear identical to other cells.

Each MSFlexgrid has properties for entering a cell and leaving a cell as well as for entering the grid and leaving the grid. Taking advantage of these properties means moving the textbox whenever we click to enter a cell and giving it the correct dimensions of that cell. When entering a cell, the textbox will take on the contents of the cell and when leaving any cell the contents of the textbox are put into that cell. As described above, the textbox will be invisible, because it will have no border and be placed directly over the cell that was clicked on. This technique allows us to change the contents of the MSFlexgrid, but it is not perfect. In order to get a value out of a textbox, another cell in the same grid must be clicked with the mouse, or the value in the textbox will be removed without being saved in the given cell. The shortcomings of the MSFlexgrid make it barely adequate as a matrix input device, but it is the only control that offers the flexibility of changing input rows and columns on the fly. There is some
complicated code connected to this procedure but it will not be discussed here because it has no connection with the transportation problem.

Having readied the MSFlexgrid for input, the program is ready to run, but at this point it can only accept input and not do anything with it. Much of the code associated with ‘Form1’ doesn’t have anything to do with the transportation problem. It is just enabling and disabling buttons and textboxes at select times to control the flow of the program. Other than flow control code, the bulk of the code in ‘Form1’ does error checking on the matrices to make sure that they contain only numbers which are positive integers.

The program flow for ‘Form1’ is quite simple. The user is presented with the form, and the program sits and waits for the user to input the company name and the quantities and names for the supply and demand points. When satisfied with the supply points, demand points, and company name, the user clicks on edit cost and capacity matrices. When clicking the ‘create cost and capacities’ button the demand quantity indicates the number of columns and the supply quantity indicates the number of rows. The modified MSFlexgrid properties draw the graph to indicate the desired number of rows and columns and begin to input. This input method will be used in three grids: one for the cost matrix, one for the supply matrix, and one for the demand matrix. The supply and demand matrixes will each have only one column, but enough rows to accommodate the required number of inputs.

Image 2 is an example of ‘Form1’ populated with data. It has 3 supply points and 4 demand points. All inputs for company name and supply and demand types and quantities and all cost, supply and demand matrices have been filled in and are ready to be put to use. After this point there will be opportunities to make changes, but essentially the data input is complete.
The program contains several forms, each with a specific purpose. We have considered, ‘Form1’, the input form. The next form is called ‘diag_form’. This form exists for only one reason. It displays the bipartite graph associated with the transportation problem. A bipartite graph is a graph split in the middle, with supply points on the left and demand points on the right, connected by lines. The lines indicate a solution to the transportation problem where each line has a cost associated with it, the cost matrix, and represents a box in the transportation tableau, discussed later. The function behind ‘diag_form’ called ‘bipartite’ answers the question how to build the bipartite graph. First look at image 3, a blank bipartite graph with three supply points and four demand points.
Because it is simply two collections of ovals and a big collection of lines, the bipartite graph is not difficult to implement. (There are a couple of additional features not visible in image 3 which will be shown later in image 4.)

When the form load function executes, it prepares the bipartite graph elements. Supply count ovals are on the left, and demand count demand ovals are on the right. The number of lines is supply count times demand count. Each oval has two different information labels, and each line is labeled with the cost of the connection it represents. An oval’s width is equal to twice its height, and its height is proportional to the height of the form. The left side of the supply ovals group is one eighth of the total width of the form from the left edge of the form. The right side of the demand ovals group is one-eighth the width of the entire form from the right edge. Lines are placed in the area between the ovals. They extend from the on the right sides of the supply ovals to the left sides of the demand ovals. Labels within
the ovals indicate supply or demand quantities. Descriptive labels are located to the left of supply ovals and to the right of demand ovals.

Image 4 shows how the bipartite graph looks after all the elements have been added to it. Notice the description in each oval and the listing of the quantity of each supply and demand point. This is the same information that was entered on ‘Form1’.

Image 4

The bipartite graph form can be displayed from either the main input form, ‘Form1’, or from the form that draws the transportation tableau, ‘tabl_form’. When drawing the bipartite graph before a solution is found, all possible lines between the two groups of ovals are drawn. Later, after the transportation tableau is drawn and a solution is found, the lines that are drawn are limited to those that represent the solution to a given transportation problem.
The Transportation Tableau

The form that draws the transportation tableau is the most important because it has the most functions related to finding the solution to a transportation problem. The transportation tableau is the focal point for this special method of solving transportation problem. A form called ‘table_form’ contains the transportation tableau.

Image 5 is a blank transportation tableau. Each cell consists of two boxes. There are a demand quantity number of cells from left to right butted right up against each other and a supply quantity of rows.

![Transportation Tableau](image)

Image 5

Each cell has a smaller inside box in the upper right hand corner. There is a label in each bigger outside box, and a textbox and label in the smaller inside box. Each label plays a part in building a solution. As the tableau coordinates are calculated for each box, each element of each box will have the coordinates calculated as well. They are placed in relation to each outside box.
Transportation Tableau Cost Edit

A feature of the program, not necessarily derived from the transportation tableau, is a convenient way to change the values of the cost matrix using the transportation tableau. Each place where the cost values are displayed, inside the inner square, also has an underlying invisible textbox which becomes visible when making changes. To use this feature, click on the ‘Tableau Edit’ menu option and click on ‘edit costs’, then either ‘edit one cost’ or ‘edit all costs’. When using the ‘edit one cost’ feature, click on a cost to change, the ‘edit all costs’ feature changes all costs. When choosing either option, a textbox takes on the value displayed in the cost label, and then changes the label’s value with the value in the textbox when the update is complete. Also changed are the cost matrix in form1, the input form, and some other background arrays.

Image 6

Image 6 shows what the transportation tableau looks like when some data have been loaded into it. The supply quantities are displayed vertically on the right side of the tableau and the demand quantities are displayed along the bottom. Because Image 6 is the transportation tableau before any solutions have
been found, there is nothing visible inside the bigger boxes. When a solution is found each cell that is a part of the solution will have a value in it indicating the number of units used for the cell. It will be important to see all these quantities along with the costs, as we will see later.

So far building the tableau simply means putting all the pieces together so that it looks like something we can use to solve problems. We haven’t actually solved anything yet.

**Balancing the Transportation Problem**

The first step on the way to solution is to make sure the transportation problem is balanced. A balanced problem is one in which the sum of the supply quantities is equal to the sum of the demand quantities. There are times where a problem is balanced without having to do any work; in those cases the balancing function is not necessary. This is most often not the case, so we need a function to check for balance and create balance if a transportation problem does not have it. This is actually quite simple. If the supply quantity sum is greater than the demand quantity sum then a dummy demand mode is created, with costs equal to zero. Nothing would actually get shipped to a dummy demand node. It is just a place holder, so it does not cost anything to have it around. Alternatively, if the demand quantity sum is greater than the supply quantity sum a dummy supply node is created, with a very large assigned cost. The prohibitive cost prevents use of a dummy supply unless it is absolutely necessary to do so.

Instead of working with quantities directly from the input form, supply quantities, demand quantities, and the cost matrix are loaded into arrays after they are inputted. The arrays help the program run faster than if the data were read from the input matrices each time they were needed. The arrays are defined when the program starts, so the values can be used without having to be reloaded. A loop calculates the sum of the demand quantities, and another calculates the sum for the supply quantities. Then the sums are compared. If they are equal, then the problem is balanced and such a notification is displayed. If the sums are not equal then the appropriate adjustments are made automatically. These adjustments include changing the arrays and the MSFlexgrid on the input form, and expanding the transportation tableau to include an additional place for supply or demand dummies.
With the problem balanced, it is time to create a solution. Recall that the basic LP introduced earlier offered more then one solution and that each solution was substituted into the objective function. There will be one optimal solution. The process of finding the optimal solution starts with a solution that may or may not be optimal. A solution to the transportation problem is called a basic feasible solution. The path to optimality begins with the initial basic feasible solution. It is tested for optimality and, if necessary, altered. When a basic feasible solution is optimal, the problem is solved.

There are three methods in the program that are used to find the initial basic feasible solution: the ‘Northwest Corner Rule’, ‘The Minimum Cost Method’, and ‘Vogel’s Rule’. The program, being an educational tool, has two options for each method of finding the initial basic feasible solution. Each method can be implemented in a step by step fashion so that what is happening from one step to the next is apparent. Push a button to see the next iteration for each method. There is also a feature to execute a whole method from the beginning, or in the middle of the step by step option.

**The Northwest Corner Rule**

The Northwest corner rule is the easiest and fastest way to find an initial basic feasible solution. Its name derives from the fact that it starts in the northwest corner of the transportation tableau. Starting in the northwest corner cell, compare the values of the supply quantity for that row with the demand quantity of that column. If demand quantity is equal to the supply quantity the program arbitrarily uses the procedure for a bigger supply quantity. If the supply quantity of the current cell is greater than the demand quantity, the current cell of the transportation tableau is given the value of the demand quantity and the supply quantity is reduced by the amount of the demand quantity. Additionally, the demand quantity is reduced to zero and the column is now unavailable to be used in the future. We say it is exhausted. If the demand quantity is greater than the supply quantity the cell is given the value of the supply quantity and the supply quantity is reduced to zero while the demand quantity is reduced by the supply quantity. The row is then exhausted and crossed out, making it unavailable for future use. Now that a row or column has been crossed out (which will be physically shown if the step by step option is used), we start again in the most Northwest corner of the adjusted matrix. The procedure described
above is repeated until there is just one cell left. The last cell of the whole transportation tableau when using the Northwest corner rule will be the southeast corner. The values of the supply and demand quantities will be equal in the last cell so that both the supply and the demand quantities get reduced to zero and the cell takes the value. Because the problem is balanced, it will happen every time that the Northwest corner rule will end up with one cell left that has an equal supply and demand quantity. A balanced problem is a stipulation of using these methods to find an initial basic feasible solution.

With an understanding of the Northwest Corner rule, next is an explanation of how the program implements it. The program uses the arrays which contain the values for the supply and demand quantities obtained from the input form. There isn’t much decision making involved when using the Northwest corner rule. It starts by using the beginning cell (1,1) as subscripts into the arrays for the supply quantities and the demand quantities compares them. The smaller value between the supply quantity and the demand quantity is placed in the label inside the outer box of the cell in the transportation tableau. Then an ‘X’ is placed in the label that displays the smaller value. The bigger value is reduced by the smaller value and updated in the label. The arrays which are used for the actual comparison are updated and an array which keeps track of which rows and columns have been used is also updated. In the step by step option a line is drawn through the row if the supply quantity is bigger and drawn through the column if the demand quantity is bigger. The quick solve method loops over and over until the array that keeps track of empty rows and columns is full. The step by step option progresses by clicks of a button to the left of the transportation tableau until the same array is full. The problem with the Northwest corner method is that it is so simple it will hardly ever produce an optimal solution.

**The Minimum Cost Method**

The minimum cost method will usually give a better solution than the Northwest corner rule. As the name suggests, the selected cells are those with the smallest costs, so the minimum cost method uses the costs matrix in addition to the supply and demand quantity matrices. The first cell to be used is the one with the smallest cost, upon finding it the procedure for finding which value to put in the cell is
identical to that of the Northwest corner rule. Locate the smallest value for supply and demand quantities and reduce the larger quantity by the smaller quantity. Then cross out the appropriate row or column and place the smaller quantity in the cell.

It should come as no surprise that the code for this procedure varies little from the code for the Northwest corner rule. There is a specially designed function called ‘step_func’ to handle the selection of the smaller quantity in the demand and supply matrices, placing the lines, placing the ‘X’ s, and filling the array which indicates which rows and columns are still open and which aren’t. The only difference in the code leading up to the ‘step_func’ function is how the cell is selected. Selecting the cell is the only difference between each of the three methods; once a cell is selected the procedure is the same. The minimum cost method code checks the cost matrix array for the smallest value and then passes it to the function which takes care of the rest.

**Vogel’s Method**

The code for selecting the cells for minimum cost method and for the Northwest corner rule is small and quick. Even though ‘Vogel’s method’ simply picks out the next appropriate cell, it is complicated. To use ‘Vogel’s method’, find the two smallest values in each row and each column then calculate the difference. Even though it is conceptually not very difficult to understand, Vogel’s method involves a lot of arithmetic and decision making. Its starts by determining if a row has been exhausted already, if yes move on. If it is still available, then it sorts the row from lowest to highest values. Next, it takes the first two values, because after sorting them they will be the two smallest, and finds the difference. It repeats the procedure for each column that has not been exhausted, column sorts it, and calculates the difference between the first two sorted values. With the differences all calculated, it finds the biggest difference for the rows and the columns. After finding this, the method takes the row or the column where the biggest difference is located and search for the smallest cost. This is the cell that gets passed to step_func next.

The code to implement this program is long. The ‘Vogel’s method’ function picks the first row that has not been marked exhausted and sorts it, then looks for the first cell in that row that is greater than
zero and calculates the row difference by subtracting the next cell with the current cell. After calculating all of the differences, the function finds the biggest one. Then the same procedures are repeated for the columns. Take the first column that has not been exhausted and sort it. Calculate the biggest difference between the smallest two values in that column by finding the first value greater than zero and subtracting it from the cell next to it. Then search for the biggest difference among the columns. Take the biggest difference for the rows and the columns and compare them. If they are equal, the code automatically favors the row, so the same procedure is followed as when the row difference is the bigger difference. Save the value for the bigger row or column and use it to search the row or column to find the smallest value in the cost matrix. When the smallest value in the row or column is found save it and combine it with the other saved value, then pass it to step_func.

**Optimality Check**

Having obtained an initial basic feasible solution, the next thing to do is find out if it is optimal. The next function checks for optimality, which starts with finding values associated with each row and column. This is also the next logical step when solving a transportation problem by hand. To check for optimality, build a new matrix based on the variables included in the current basic feasible solution. The new matrix will be called the adjusted cost matrix, and it will indicate if the current solution is optimal. If the current solution is not feasible, the adjusted cost matrix will indicate which variable should enter the basic feasible solution. The values for each cell in the current solution will be zero in the adjusted cost matrix, and all other cells that are not part of the current basic feasible solution will have adjusted values that could be positive, negative, or zero. When all the values in the adjusted cost matrix are less than or equal to zero, then the solution is optimal. If the adjusted cost matrix has a value greater than zero it is suboptimal. The cell with the greatest value in the adjusted cost matrix is entered into the next basic feasible solution by a procedure that will be discussed later. The code for finding the values associated with each row and column is quite difficult to implement. When finding the values, by convention, assign row 1 a value of zero and look for connections between rows and columns, variables in the current solution to relate a row to a column. These connections will constitute a series of equations that will
provide the respective values for the rows and the columns. The loop for this code was not set to a specific number of iterations like all the other loops; this loop continues until all solutions are found, which makes it pretty inefficient. It makes several passes by starting with the first row associated value being zero and looking for connections between rows and columns. Each pass may only find one or two connections; when the loop starts a new pass it uses updated information to look for new connections until all connections have been found. Once all row and column associated values have been found the adjusted cost matrix can be calculated. Each cell’s adjusted cost matrix value is calculated by taking the associated row and column values, adding them together, and then subtracting that value from cost value found in the regular cost matrix. When all the adjusted cost matrix values have been calculated then a solution can be checked for optimality. The code simply seeks out the largest value over zero, if it finds one then it is suboptimal; otherwise it is optimal.

Image 7 is the fourth form which is displayed at the same time as table_form, the transportation tableau, and Form1, the input form. The fourth form, called ‘Form2,’ shows the MSFlexgrids, which contain the associated row and column values, the adjusted cost matrix, and the current solution. As shown below the associated row and column values, labeled ‘u’ and ‘v’ respectively, are in the top left corner. The adjusted cost matrix is below it and below that are some labels that indicate if the current solution is optimal and what the z value is.
The objective function value, called the z-value, is simply the sum of the cost of each variable taken from the cost matrix cost multiplied by the value of the variable in the current solution transportation tableau. There is little code behind the fourth form, most of the code is run by tabl_form and put onto ‘Form2.’

What if the current solution is not optimal? Then the following procedure to change the current basic feasible solution by removing a variable and entering a different variable is to be followed. After one variable has been replaced by another, the whole procedure to check for optimality is repeated. The method for changing a basic feasible solution is called the transportation simplex.

The Transportation Simplex

The transportation simplex begins by determining which variable is to be added. This idea is already very familiar, because a check of the adjusted cost matrix indicates which variable should be
entered into the basic feasible solution. Knowing which variable to enter is the first step, the next is to
determine which variables will be affected by the entering variable. This process is accomplished by
drawing a loop. But not just any loop will reveal the correct information. Image 8 shows a basic feasible
solution using the ‘Minimum Cost Method’ starting with the same example that has been used throughout
for the screen shots from the program. The basic feasible solution variables are the cells that have the
number in the bigger box. Cell (1, 3) is the next cell to enter the basic feasible solution. Its adjusted cost
matrix value for this problem has a value of 4400. Interestingly, ‘Vogel’s method’ gives an optimal
solution for this problem. With cell (1, 3) entering the basic feasible solution, draw the loop as illustrated
in Image 8.
When drawing the loop connections can only be horizontal or vertical, only two connections may be made in any one row or column, and the first cell should be in common with the last cell. A property of the loop is that there can be only one loop. Also, a loop cannot be drawn over a basic feasible solution. Having found the loop, identify which variables in the loop are even and which are odd with respect to the starting point of the loop. Cells that are labeled as even cells will decrease in value by the cell with the smallest value and cells that are odd will increase by that same amount. After performing these calculations one of the old basic feasible solution variables will be reduced to zero and therefore removed from the basic feasible solution, and the entering variable will now have a value in the new solution. The loop is fundamental when finding out which variables are going to be part of the new basic feasible solution after determining which non-basic variable is going to become basic. When finished calculating the new solution, repeat the optimality check and see if the new solution is optimal. If it is not, then figure out which variable to enter and repeat the process until an optimal solution is found. The code for this final algorithm was difficult to write. There is no easy way to construct the loop, so the concept of a spanning tree came into play. The first thing the function that takes care of the transportation simplex does is to build a spanning tree based on a basic feasible solution. When building the spanning tree, the function starts by looking for the row with the most variables in it, this is level zero. Then those variables are connected to the first node in level 1 and called level 2. From level two the function imitates the code that searched for the associated row and column values as part of the optimality function. The code starts at the beginning of the tableau and checks cells that are part of the current solution are that are not already connected; it searches for connections in a loop until all connections are made. Because VB6 is not setup to handle things like trees very well, the spanning tree was sloppily assembled using an array of strings. The indexes to the strings worked handily as indicators of the levels, but parent nodes were placed physically above child nodes to indicate connections between them. Despite being improvised, searching the spanning tree for each node was not difficult because of the built in string function in VB6. In order to determine which variables in the current solution would be included in the loop the following algorithm was implemented. First determine the cell to enter the solution, and then search for each node
in the spanning tree. If the two nodes were not on the same level then get the parent node of the lower level node. The two pieces of information, the child node and the parent node, together indicate a cell that is part of the loop. Next check to see if the two nodes are on the same level, when they are, find their parent node and use the parent-child node connections to finish determining which variables are included in the loop. Having laid out all the points in the loop the next algorithm draws the lines to connect the loop and at the same time determines whether each cell is even or odd. Start with the entering cell look left and right for the next connection; when found draw then line and label as an odd cell. Now, from the second cell look up and down for the next connection, make the connection, and label the cell even. Continue this pattern until connected up with the first cell, drawing lines and alternately labeling each cell as even or odd. With all the cells labeled even and odd it is easy to finish the simplex algorithm. Find the smallest value in the loop, do a search going through all the cells in the loop starting from the beginning, then add the smallest cell’s value to the even cell and subtract it from the odd cells. When this is finished call the function ‘optimal’ and it will check for optimality. If the new solution is not optimal then run the simplex algorithm again.

That's it! The transportation problem is solved. Now that we have solved the transportation problem there is one last thing to discuss. There is a subset problem of the transportation problem call the ‘Assignment problem.’ The assignment problem is setup the same as the transportation problem with two changes to the properties. The sum of all the supply constraints and demand constraints is 1, and there is an associated cost matrix but it is a square matrix. The assignment problem can represent a real world situation where there are n number of jobs and n number of people to fill them. The cost matrix indicates each person’s proficiency for each job so the optimal solution represents the best way to assign n people to do n jobs.

Solving the transportation problem is complicated. When first covering the material it is confusing and difficult to understand exactly how to develop an initial basic feasible solution. Even harder to understand is the transportation simplex. If done incorrectly you can swap variables in an out all day without ever finding an optimal solution. This program removes the questions that come with
learning about the transportation problem. Following along is easy and verifying that arithmetic was handled correctly is fast. This program can handle a transportation problem with a cost matrix of 50 by 50, which represents a huge problem and to do it by hand would take a long time. Even though it is graphically based most of the calculations take place behind the scenes, so with some slight modification the program could handle a cost matrix of hundreds by hundreds. Compared with commercial problem solver the data entry is more forgiving and is easier to see and correct due to the graphical nature of the input grids. The transportation problem has numerous applications in business and science, specifically computers, so understanding it is vital to being able to move forward in these areas.
References

Appendix A

Program Flowcharts

Form1 flowchart . . . . . . . . . A-2
Balance . . . . . . . . . . . . . . . A-3
Northwest Corner Rule flowchart . . . . . . . . . . . . . . . A-4
Minimum Cost Method flowchart . . . . . . . . . . . . . . . A-5
Vogel’s Method flowchart . . . . . . . . . . . . . . . . . . . A-6
Program Start

Input Supply and Demand names and quantities

Create cost matrix and supply and demand quantity

Error check input matrices

Edit matrices?

Show Bipartite graph?

Save current settings?

Load settings from file?

Load settings

Load diag_form

Display Bipartite Graph

Save settings

Load tabl_form

End Program
Balance Start

Sum Demand quantities

Sum Supply quantities

Quantities equal?

Balanced

Supply quantity sum

Add a demand quantity and set to difference

Balanced

Add a supply quantity and set to difference

Balanced
Start Northwest Corner Rule

In southwest corner?

Select Most northwest corner cell

Demand quantity = Supply quantity?

Demand quantity > Supply quantity?

Set Cell value to Supply quantity

Set Cell value to Supply quantity

Subtract supply quantity from

Subtract supply quantity from

Set row to exhausted

Set row to exhausted

Done
Start Minimum Cost Method

All cells exhausted?

Select cell with smallest cost

Demand quantity = Supply quantity?

Demand quantity > Supply quantity?

Set Cell value to Supply quantity

Set row to exhausted

Subtract supply quantity from demand

Set row to exhausted

Subtract supply quantity from demand

Set row to exhausted

Demand quantity > Supply quantity?

Set Cell value to Supply quantity

Done
Start Vogel's Method

1. Find difference between two smallest cells for each row

2. Pick largest difference

3. Select cell with smallest cost in row or column with largest difference

4. Demand quantity = Supply quantity?

5. If yes, set Cell value to Supply quantity; if no, proceed to next step.

6. Demand quantity > Supply quantity?

7. If yes, subtract supply quantity from demand quantity and set row to exhausted;

8. If no, set Cell value to Supply quantity and subtract supply quantity from demand quantity, then set row to exhausted.

Done
# Appendix B

## Selected functions from the program

<table>
<thead>
<tr>
<th>Function Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Verification</td>
<td>B-2</td>
</tr>
<tr>
<td>File Save</td>
<td>B-3</td>
</tr>
<tr>
<td>File Load</td>
<td>B-5</td>
</tr>
<tr>
<td>Generate bipartite graph</td>
<td>B-8</td>
</tr>
<tr>
<td>Create transportation tableau</td>
<td>B-10</td>
</tr>
<tr>
<td>Balance check</td>
<td>B-13</td>
</tr>
<tr>
<td>Balance</td>
<td>B-14</td>
</tr>
<tr>
<td>Quick solve for Northwest Corner Rule</td>
<td>B-15</td>
</tr>
<tr>
<td>Quick solve for Minimum Cost Method</td>
<td>B-16</td>
</tr>
<tr>
<td>Quick solve for Vogel’s Method</td>
<td>B-16</td>
</tr>
<tr>
<td>Step_func</td>
<td>B-20</td>
</tr>
<tr>
<td>Check for optimality</td>
<td>B-21</td>
</tr>
</tbody>
</table>
The following code is from Form One. This code verifies the values in each of the input matrices, one for the cost matrix and one for the supply and demand quantity matrices. If all the values of the matrices are acceptable then the values are loaded into arrays. The program uses the values from the arrays to execute the subsequent functions of the program, not the input matrices. Next, the function adds up the values of the supply quantities and the demand quantities and compares them. Lastly the function compares the sums of the supply and demand quantities see if the problem is balanced.

```
Private Sub Command4_Click()
Dim x As Long
Dim y As Long
Dim count As Long
ReDim cost_matrix(sup_cnt, dmd_cnt) As Long
ReDim dmd_cap(dmd_cnt) As Long
ReDim sup_cap(sup_cnt) As Long

Command3.Caption = "Change Cost Matrix"
Command3.Visible = True
Command4.Visible = False
Command1.Enabled = True
Command2.Enabled = True
mnu_save.Enabled = True

MSFlexGrid(1).Enabled = False
MSFlexGrid(2).Enabled = False
MSFlexGrid(3).Enabled = False

Text6.Visible = False
For y = 1 To dmd_cnt
    If IsNumeric(MSFlexGrid(3).TextMatrix(y, 1)) = True Then
        dmd_cap(y) = MSFlexGrid(3).TextMatrix(y, 1)
    Else
        dmd_cap(y) = -1
    End If
End For

For x = 1 To sup_cnt
    If y = 1 Then
        If IsNumeric(MSFlexGrid(2).TextMatrix(x, 1)) = True Then
            sup_cap(x) = MSFlexGrid(2).TextMatrix(x, 1)
        Else
            sup_cap(x) = -1
        End If
    End If
End For

If IsNumeric(MSFlexGrid(1).TextMatrix(x, y)) = True Then
    cost_matrix(x, y) = MSFlexGrid(1).TextMatrix(x, y)
Else
    cost_matrix(x, y) = -1
End If
```
```
Next x
Next y

count = 0
For x = 1 To sup_cnt
    If IsNumeric(MSFlexGrid(2).TextMatrix(x, 1)) = True Then
        count = count + MSFlexGrid(2).TextMatrix(x, 1)
    Else
        count = 0
        x = sup_cnt
    End If
Next x
If count = 0 Then
    Label11.Caption = "Alphas"
Else
    Label11.Caption = count
End If

count = 0
For x = 1 To dmd_cnt
    If IsNumeric(MSFlexGrid(3).TextMatrix(x, 1)) = True Then
        count = count + MSFlexGrid(3).TextMatrix(x, 1)
    Else
        count = 0
        x = dmd_cnt
    End If
Next x
If count = 0 Then
    Label12.Caption = "Alphas"
Else
    Label12.Caption = count
End If

    Label9.Caption = "Balanced"
Else
    Label9.Caption = "Not Balanced"
End If

End Sub
```

Form One also had a load and save feature that could load and save problems to prevent from have to repeatedly type in the same problem. The following is the code for the save feature. The save feature first open the common save dialog box, that is common to all Microsoft programs, and asks to provide a file name. The function then takes the file name, opens the file, and proceeds to write various pieces of the data from the transportation problem to the file.

```
Private Sub mnu_save_Click()
Dim fileoutput As String
Dim x As Integer
```
Dim y As Integer

CommonDialog1.Filter = "Operations Research Files (*.orf)|*.orf"
CommonDialog1.Flags = cdlOFNOverwritePrompt Or cdlOFNNoChangeDir
CommonDialog1DialogTitle = "Save Operations Research Files"
CommonDialog1.ShowSave

If CommonDialog1.FileName <> "" Then
  Open CommonDialog1.FileName For Output As #1

  Write #1, "[Company Name]"
  Write #1, Text5.Text

  Write #1, "[Supply Type]"
  Write #1, Text1.Text

  Write #1, "[Supply Quantity]"
  Write #1, Text2.Text

  Write #1, "[Demand Type]"
  Write #1, Text3.Text

  Write #1, "[Demand Quantity]"
  Write #1, Text4.Text

  Write #1, "[Cost Matrix]"

  For x = 1 To sup_cnt
    fileoutput = ">"
    For y = 1 To dmd_cnt
      fileoutput = fileoutput + CStr(MSFlexGrid(1).TextMatrix(x, y))
      If y <> dmd_cnt Then
        fileoutput = fileoutput + ","
      End If
    Next y
  Next x

  Write #1, fileoutput

  Write #1, "[Supplier Capacity]"
  fileoutput ="
  For x = 1 To sup_cnt
    fileoutput = fileoutput + MSFlexGrid(2).TextMatrix(x, 1)
    If x <> sup_cnt Then
      fileoutput = fileoutput + ","
    End If
  Next x
  Write #1, fileoutput

  Write #1, "[Demand Capacity]"
  fileoutput ="
  For x = 1 To dmd_cnt
    fileoutput = fileoutput + MSFlexGrid(3).TextMatrix(x, 1)
    If x <> dmd_cnt Then
      fileoutput = fileoutput + ","
    End If
  Next x
  Write #1, fileoutput
The last important function from Form One is the Load function. This function displays the common file open dialog box and asks for a file name. It checks to make sure the file name is valid and if the user wants to overwrite the file if it exists. Then the function reads the file line by line and compares the section headings in the file to find and read in pieces of the transportation problem.

Private Sub mnu_load_Click()
    Dim fileinput As String
    Dim x, y, u As Integer
    Dim stemp As String
    Dim count As Long
    Dim go As Integer

    If Command3.Caption = "Change Cost Matrix" Then
        If MsgBox("Overwrite existing LP?", 4) = 6 Then
            go = 1
        End If
    Else
        go = 1
    End If

    If go = 1 Then
        CommonDialog1.Filter = "Operations Research Files (*.orf)\/*.orf"
        CommonDialog1.DialogTitle = "Open Operations Research Files"
        CommonDialog1.Flags = cdlOFNHideReadOnly Or cdlOFNNoChangeDir
        CommonDialog1.ShowOpen
        If CommonDialog1.FileName <> "" Then
            Open CommonDialog1.FileName For Input As #1
            Do Until EOF(1)
                Input #1, fileinput
                Select Case (fileinput)
                    Case "[Company Name]"
                        Input #1, fileinput
                        Text5.Text = fileinput
                    Case "[Supply Type]"
                        Input #1, fileinput
                        Text1.Text = fileinput
                    Case "[Supply Quantity]"
                        Input #1, fileinput
                        sup_cnt = CInt(fileinput)
                End Select
            Loop
        End If
    End If
End Sub
MSFlexGrid(1).Rows = sup_cnt + 1
MSFlexGrid(2).Rows = sup_cnt + 1
Text2.Text = sup_cnt

Case "[Demand Type]"
Input #1, fileinput
Text3.Text = fileinput

Case "[Demand Quantity]"
Input #1, fileinput
dmd_cnt = CInt(fileinput)
MSFlexGrid(1).Cols = dmd_cnt + 1
MSFlexGrid(3).Rows = dmd_cnt + 1
Text4.Text = dmd_cnt

Case "[Cost Matrix]"
For y = 1 To sup_cnt
Input #1, fileinput
u = 0
stemp = ""
For x = 2 To Len(fileinput)
If x <> Len(fileinput) Then
    If Mid(fileinput, x, 1) = "," Then
        u = u + 1
        MSFlexGrid(1).TextMatrix(y, u) = stemp
        stemp = ""
    Else
        stemp = stemp + Mid(fileinput, x, 1)
    End If
Else
    u = u + 1
    stemp = stemp + Mid(fileinput, x, 1)
    MSFlexGrid(1).TextMatrix(y, u) = stemp
End If
Else
    u = u + 1
    stemp = stemp + Mid(fileinput, x, 1)
End If
Next x
Next y

Case "[Supplier Capacity]"
Input #1, fileinput
u = 0
y = 1
stemp = ""
For x = 1 To Len(fileinput)
If x <> Len(fileinput) Then
    If Mid(fileinput, x, 1) = "," Then
        u = u + 1
        MSFlexGrid(2).TextMatrix(u, y) = stemp
        stemp = ""
    Else
        stemp = stemp + Mid(fileinput, x, 1)
    End If
Else
    u = u + 1
    stemp = stemp + Mid(fileinput, x, 1)
End If
Else
    u = u + 1
    stemp = stemp + Mid(fileinput, x, 1)
End If
Next x
Case "[Demand Capacity]"
    Input #1, fileinput
    u = 0
    y = 1
    stemp = ""
    For x = 1 To Len(fileinput)
        If x <> Len(fileinput) Then
            If Mid(fileinput, x, 1) = "," Then
                u = u + 1
                MSFlexGrid(3).TextMatrix(u, y) = stemp
                stemp = ""
            Else
                stemp = stemp + Mid(fileinput, x, 1)
            End If
        Else
            u = u + 1
            stemp = stemp + Mid(fileinput, x, 1)
            MSFlexGrid(3).TextMatrix(u, y) = stemp
        End If
    Next x
    End Select
Loop
Close #1

For x = 1 To sup_cnt
    MSFlexGrid(1).TextMatrix(x, 0) = Text1.Text + " " + CStr(x)
    MSFlexGrid(2).TextMatrix(x, 0) = Text1.Text + " " + CStr(x)
Next x

For x = 1 To dmd_cnt
    MSFlexGrid(3).TextMatrix(x, 0) = Text3.Text + " " + CStr(x)
    MSFlexGrid(1).TextMatrix(0, x) = Text3.Text + " " + CStr(x)
Next x

Text1.Enabled = False
Text3.Enabled = False
Text2.Enabled = False
Text4.Enabled = False
Text5.Enabled = False
Call Command4_Click
End If
End If
End Sub

The code for generating the bi partite graph is alone in the next form, called diag_form. In order to generate this graph the function uses several controls that are loaded onto the form when the form loads. In particular the ovals and the lines are the important parts of the graph because they represent the supply and demand points. The lines represent the connections between each supply and demand and its cost. To draw the graph the function first relates the size of each
oval to the size of the form and places the supply point ovals on the left a distance from the left edge of the form. Then
the demand point ovals are place a distance from the right edge of the form. Next, the lines are drawn and the costs are
added. If the user has found a basic feasible solution then only the lines and costs that represent that solution are drawn.

Private Sub bipartite()

    Dim cnt As Long
    Dim x, y, z As Long
    Dim m As Double

    Dim sup_ystart As Long
    Dim sup_xstart As Long
    Dim sup_inc As Long

    Dim dmd_ystart As Long
    Dim dmd_xstart As Long
    Dim dmd_inc As Long

    sup_ystart = diag_form.ScaleHeight / (1.5 * sup_cnt + 1.5)
    sup_xstart = diag_form.ScaleWidth / 8
    sup_inc = sup_ystart

    dmd_ystart = diag_form.ScaleHeight / (1.5 * dmd_cnt + 1.5)
    dmd_xstart = diag_form.ScaleWidth - diag_form.ScaleWidth / 8
    dmd_inc = dmd_ystart

    For x = 1 To sup_cnt
        With sup(x)
            .Top = sup_ystart
            .Height = sup_inc
            .Width = .Height * 2
            .Left = sup_xstart
            .Visible = True
        End With

        With sup_lab(x)
            .Left = sup(x).Left + sup(x).Width / 4
            .Top = sup(x).Top + sup(x).Height / 3
            .Caption = sup_type + " " + CStr(x)
            .Visible = True
        End With

        With sup_s(x)
            .Left = sup(x).Left - .Width
            .Top = sup(x).Top + sup(x).Height / 3
            .Caption = "s" + CStr(x) + " = " + CStr(sup_cap(x))
            .Visible = True
        End With

        sup_ystart = sup_ystart + 1.5 * sup_inc
    Next x
For y = 1 To dmd_cnt
    With dmd(y)
        .Top = dmd_ystart
        .Height = dmd_inc
        .Width = .Height * 2
        .Left = dmd_xstart - .Width
        .Visible = True
    End With

    With dmd_lab(y)
        .Left = dmd(y).Left + dmd(y).Width / 4
        .Top = dmd(y).Top + dmd(y).Height / 3
        .Caption = dmd_type + " " + CStr(y)
        .Visible = True
    End With

    With dmd_d(y)
        .Left = dmd(y).Left + dmd(y).Width + 150
        .Top = dmd(y).Top + dmd(y).Height / 3
        .Caption = "d" + CStr(y) + " = " + CStr(dmd_cap(y))
        .Visible = True
    End With

    dmd_ystart = dmd_ystart + 1.5 * dmd_inc
End For

If sup_dmy = 1 Then
    sup_lab(sup_cnt) = "Supply Dummy"
    sup_s(sup_cnt) = "SD = " + CStr(sup_cap(sup_cnt))
End If

If dmd_dmy = 1 Then
    dmd_lab(dmd_cnt) = "Demand Dummy"
    dmd_d(dmd_cnt) = "DD = " + CStr(dmd_cap(dmd_cnt))
End If

z = 0
If ibfs_stat = 0 Then
    For x = 1 To sup_cnt
        For y = 1 To dmd_cnt
            z = z + 1

            With Line1(z)
                .X1 = sup(x).Left + sup(x).Width
                .X2 = dmd(y).Left
                .Y1 = sup(x).Top + sup(x).Height / 2
                .Y2 = dmd(y).Top + dmd(y).Height / 2
                .Visible = True
            End With

            With cost(z)
                .Left = Line1(z).X1 + ((Line1(z).X2 - Line1(z).X1) / 5)
                .Top = Line1(z).Y1 + ((Line1(z).Y2 - Line1(z).Y1) / 5) - (.Height + 10)
                .Caption = cost_matrix(Int((z - 1) / (dmd_cnt)) + 1, ((z - 1) Mod (dmd_cnt)) + 1)
                .Visible = True
            End With
        Next y
    Next x
End If
The last form that has any valuable code in it is called tabl_form. Tabl_form is the form where most of the work is done for the transportation problem including balancing the problem, finding an initial basic feasible solution, checking for optimality, and using the transportation simplex. The following code is to build and display the transportation tableau.

The function first determines the size of the outside box in relation to the size of the form and the size of the inside box in relation to the size of the outside box. Then textboxes and labels are placed in the inside and outside boxes to display information; this process is repeated supply count time demand count number of times. Lastly the supply quantities are displayed vertically on the right side of the transportation tableau and the demand quantities are display along the bottom.

Private Sub create_tableau()

Dim x, y, z As Long
Dim left_start  As Long
Dim top_start   As Long
Dim box_width   As Long
Dim box_height  As Long
Dim xoutr_inc   As Long

End Sub
Dim youtr_inc As Long

box_width = (tabl_form.ScaleWidth) / (dmd_cnt + 1)
box_height = (tabl_form.ScaleHeight) / (sup_cnt + 1)

z = 0
top_start = box_height / 2
For y = 1 To sup_cnt
    left_start = box_width / 2
    For x = 1 To dmd_cnt
        z = z + 1
        With outer(z)
            .Left = left_start
            .Top = top_start
            .Width = box_width
            .Height = box_height
            .Visible = True
        End With

        With inner(z)
            .Left = outer(z).Left + outer(z).Width / 2
            .Width = outer(z).Width / 2
            .Height = outer(z).Height / 2
            .Top = outer(z).Top
            .Visible = True
        End With

        With cost_lbl(z)
            .Left = inner(z).Left + inner(z).Width / 8
            .Top = inner(z).Top + inner(z).Height / 4
            .Width = 3 * inner(z).Width / 4
            .Visible = True
        End With

        With var_lbl(z)
            .Left = outer(z).Left + box_width / 8
            .Top = outer(z).Top + (2 * box_height / 3)
            .Width = 3 * outer(z).Width / 4
            .Visible = True
        End With

        With cost_txt(z)
            .Left = inner(z).Left + inner(z).Width / 4
            .Top = inner(z).Top + inner(z).Height / 4
            .Visible = False
        End With

        With var_txt(z)
            .Left = outer(z).Left + box_width / 4
            .Top = outer(z).Top + (2 * box_height / 3)
            .Visible = False
        End With

        With line_points(z)
            .Left = outer(z).Left + 3 * box_width / 8
        End With
    Next x
    left_start = box_width / 2
Next y
With eo_label(z)
    .Left = outer(z).Left + 6 * box_width / 8
    .Top = outer(z).Top + 12 * box_height / 16
    .Visible = True
End With

looper(z).Visible = False

If z <= dmd_cnt Then
    With vconnect_line(z)
        .X1 = outer(z).Left + (3 * outer(z).Width / 8) + line_points(z).Width / 2
        .X2 = outer(z).Left + (3 * outer(z).Width / 8) + line_points(z).Width / 2
        .Y1 = outer(z).Top
        .Y2 = (outer(z).Height * sup_cnt) + outer(z).Top
        .Visible = False
    End With
End If

If y = sup_cnt Then
    With dmd_lbl(z - (sup_cnt - 1) * dmd_cnt)
        .Top = outer(z).Top + outer(z).Height + 150
        .Width = 3 * outer(z).Width / 4
        .Left = outer(z).Left + outer(z).Width / 3
        .Visible = True
    End With
End If

If create = True Then
    dmd_lbl(z - (sup_cnt - 1) * dmd_cnt).Caption = dmd_cap(x)
End If

End If

If create = True Then
    cost_lbl(z).Caption = cost_matrix(y, x)
    var_lbl(z).Caption = ""
    cost_txt(z).Text = ""
    var_txt(z).Text = ""
    line_points(z).Visible = False
    eo_label(z).Visible = False
End If

left_start = left_start + box_width
Next x

With hconnect_line(y)
    .X1 = outer(z + 1 - dmd_cnt).Left
    .X2 = outer(z + 1 - dmd_cnt).Left + (outer(z + 1 - dmd_cnt).Width * dmd_cnt)
    .Y1 = outer(z).Top + 9 * outer(z).Height / 16
    .Y2 = outer(z).Top + 9 * outer(z).Height / 16
    .Visible = False
End With

With sup_lbl(y)
If create = True Then
    sup_lbl(y).Caption = sup_cap(y)
End If

Next y
create = False
End Sub

The next piece of code is a function that follows logically in the solution to the transportation problem, making sure the problem is balanced. There is a menu option called balance that works in two pieces, the first checks to see if the problem is balanced and the second balances the problem if it is unbalanced. The first function called bal_check sums the quantities for supply and demand. Then it compares the two and returns zero if they are equal. If they are not equal the next function will be executed. The first action taken in balance is to create an extra demand quantity or supply quantity in each of the arrays; then the function indicates that the problem is not balanced and a dummy exists. Finally the function indicates the program is now balanced.

Function bal_check() As Integer
Dim x As Integer
Dim y As Integer
counta = 0
countb = 0

For x = 1 To dmd_cnt
    If IsNumeric(dmd_cap(x)) = True Then
        countb = countb + dmd_cap(x)
    Else
        countb = 0
        x = dmd_cnt
    End If
Next x

For x = 1 To sup_cnt
    If IsNumeric(sup_cap(x)) = True Then
        counta = counta + sup_cap(x)
    Else
        counta = 0
        x = sup_cnt
    End If
Next x
End If
Next x

If counta = countb Then
    bal_check = 0
Else
    If counta > countb Then
        bal_check = 1
    Else
        bal_check = 2
    End If
End If

End Function

Function balance() As Integer
Dim x As Integer
Dim y As Integer
Dim z As Integer
Dim temp() As Long

z = bal_check

If z <> 0 Then
    balance = 0
    Form1.Label9.Caption = "Balanced"
    If bal_check = 1 Then
        dmd_dmy = 1
        sup_dmy = 0
        ReDim temp(dmd_cnt)
        For x = 1 To dmd_cnt
            temp(x) = dmd_cap(x)
        Next x
        ReDim dmd_cap(dmd_cnt + 1)
        For x = 1 To dmd_cnt
            dmd_cap(x) = temp(x)
        Next x
        dmd_cnt = dmd_cnt + 1
        dmd_cap(dmd_cnt) = counta - countb
    Else
        sup_dmy = 1
        dmd_dmy = 0
        ReDim temp(sup_cnt)
        For x = 1 To sup_cnt
            temp(x) = sup_cap(x)
        Next x
        ReDim sup_cap(sup_cnt + 1)
        For x = 1 To sup_cnt
            sup_cap(x) = temp(x)
        Next x
    End If
End If
sup_cnt = sup_cnt + 1
sup_cap(sup_cnt) = countb - counta
End If

ReDim temp(sup_cnt - sup_dmy, dmd_cnt - dmd_dmy)
For x = 1 To dmd_cnt - dmd_dmy
  For y = 1 To sup_cnt - sup_dmy
    temp(y, x) = cost_matrix(y, x)
  Next y
Next x

ReDim cost_matrix(sup_cnt, dmd_cnt)
For x = 1 To dmd_cnt
  For y = 1 To sup_cnt
    cost_matrix(y, x) = 0
  Next y
Next x

For x = 1 To dmd_cnt - dmd_dmy
  For y = 1 To sup_cnt - sup_dmy
    cost_matrix(y, x) = temp(y, x)
  Next y
Next x
Else
  balance = 1
End If
End Function

The next function to look at is the quick solve option for creating an initial basic feasible solution with the Northwest Corner rule. There isn’t much to this function because the Northwest Corner Rule is conceptually very simple. This function simply tests each row left to right, if that row and column has not been exhausted then it is next for step_func, which takes care of deciding how to proceed; step_func will be described later.

Case "NWC Step"
  For y = 1 To sup_cnt
    For x = 1 To dmd_cnt
      If ibfs(y, x) > -2 Then
        Call step_func(x, y)
        x = dmd_cnt
        y = sup_cnt
      End If
    Next x
  Next y

The minimum cost method is the next function to look at. This function go through the cost matrix row by row from left to right looking for the smallest cost in a row or column that has not been exhausted. When the next value is found it is passed to step_func.
Case "MCM Step"
For y = 1 To sup_cnt
    For x = 1 To dmd_cnt
        If ibfs(y, x) > -2 Then
            minim = cost_matrix(y, x)
            ysave = y
            xsave = x
            x = dmd_cnt
            y = sup_cnt
        End If
    Next x
Next y

For y = 1 To sup_cnt
    For x = 1 To dmd_cnt
        If ibfs(y, x) > -2 And cost_matrix(y, x) < minim Then
            minim = cost_matrix(y, x)
            ysave = y
            xsave = x
        End If
    Next x
Next y

If xsave > 0 And ysave > 0 Then
    Call step_func(xsave, ysave)
End If

The most complicated of the three initial basic feasible solution methods is Vogel’s rule; this function is long.
The first step in the function is to sort each row and then pick the first two sorted cells as the smallest two elements. Next sort the columns and pick the first two sorted cells as the smallest two elements. Now, find the difference between the smallest two elements for each row and each column. Then look for the largest element in each row and each column, compare the two largest elements to find the larger of the two elements. This finds the largest difference. Then take that row or column and search it to find the smallest cost in that row or column. This is the cell that is passed to step_func.

Case "Vogel's Step"
Dim x As Long
Dim y As Long
Dim z As Long
Dim xsave As Long
Dim ysave As Long
Dim minim As Long
Dim row_min() As Long
Dim col_min() As Long
Dim nf _ As Boolean
ReDim row_min(sup_cnt)
ReDim col_min(dmd_cnt)

For y = 1 To sup_cnt
    row_min(y) = -2
Next y

For x = 1 To dmd_cnt
    col_min(x) = -2
Next x

ReDim passr(dmd_cnt)

For y = 1 To sup_cnt
    For x = 1 To dmd_cnt
        If ibfs(y, x) > -2 Then
            passr(x) = cost_matrix(y, x)
        Else
            passr(x) = -1
        End If
    Next x

Call vogAst(dmd_cnt)

nf = False
For z = 1 To dmd_cnt
    If passr(z) >= 0 Then
        Exit For
    Else
        If z = dmd_cnt Then
            nf = True
        End If
    End If
Next z

If nf = False Then
    If z = dmd_cnt Then
        row_min(y) = passr(z)
    Else
        row_min(y) = passr(z + 1) - passr(z)
    End If
Else
    row_min(y) = -1
End If
Next y

For z = 1 To sup_cnt
    If row_min(z) >= 0 Then
        minim = row_min(z)
        ysave = z
        Exit For
    End If
Next z

For y = z To sup_cnt
    If row_min(y) > minim And row_min(y) >= 0 Then
        ysave = y
        minim = row_min(y)
    End If
Next y

'sort cols second
ReDim passr(sup_cnt)
For x = 1 To dmd_cnt
    For y = 1 To sup_cnt
        If ibfs(y, x) > -2 Then
            passr(y) = cost_matrix(y, x)
        Else
            passr(y) = -1
        End If
    Next y
Call vog_ast(sup_cnt)

nf = False
For z = 1 To sup_cnt
    If passr(z) >= 0 Then
        Exit For
    Else
        If z = sup_cnt Then
            nf = True
        End If
    End If
Next z

If nf = False Then
    If z = sup_cnt Then
        col_min(x) = passr(z)
    Else
        col_min(x) = passr(z + 1) - passr(z)
    End If
Else
    col_min(x) = -1
End If
Next x

For z = 1 To dmd_cnt
    If col_min(z) >= 0 Then
        minim = col_min(z)
        xsave = z
        Exit For
    End If
Next z

For x = z To dmd_cnt
    If col_min(x) > minim And col_min(x) >= 0 Then
        xsave = x
        minim = col_min(x)
    End If
Next x

If col_min(xsave) = row_min(ysave) Then
    For z = 1 To sup_cnt
        If ibfs(z, xsave) > -2 Then
            minim = cost_matrix(z, xsave)
            ysave = z
            Exit For
        End If
    Next z
End If
End If
Next z

For y = z To sup_cnt
  If cost_matrix(y, xsave) < minim And ibfs(z, xsave) > -2 Then
    minim = cost_matrix(y, xsave)
    ysave = y
  End If
Next y
Else
  If col_min(xsave) > row_min(ysave) Then
    For z = 1 To sup_cnt
      If ibfs(z, xsave) > -2 Then
        minim = cost_matrix(z, xsave)
        ysave = z
        Exit For
      End If
    Next z

    For y = z To sup_cnt
      If cost_matrix(y, xsave) < minim And ibfs(z, xsave) > -2 Then
        minim = cost_matrix(y, xsave)
        ysave = y
      End If
    Next y
  Else
    For z = 1 To sup_cnt
      If ibfs(ysave, z) > -2 Then
        minim = cost_matrix(ysave, z)
        xsave = z
        Exit For
      End If
    Next z

    For x = z To dmd_cnt
      If cost_matrix(ysave, x) < minim And ibfs(z, xsave) > -2 Then
        minim = cost_matrix(ysave, x)
        xsave = x
      End If
    Next x
  End If
End If
End If

If xsave > 0 And ysave > 0 Then
  Call step_func(xsave, ysave)
End If

What does step_func do? Here it is. Step_func takes two values, a row and a column that indicate a specific cell. The first thing this function does is to compare the demand quantity with the supply quantity. If they are equal then the function follows the procedure as if the demand quantity was bigger. If the demand quantity is bigger than the supply quantity the function puts the supply quantity value in the cell, crosses out the supply quantity, and reduces the bigger
quantity by the supply quantity. If the demand quantity is smaller the procedure is the same but with the quantities reversed.

Private Function step_func(x As Long, y As Long)
Dim z As Long

If dmd_tmp(x) = sup_tmp(y) Then
    bfs(y, x) = sup_tmp(y)
    var_lbl((y - 1) * (dmd_cnt) + x) = bfs(y, x)
    sup_lbl(y) = "X"
    dmd_lbl(x) = dmd_tmp(x) - sup_tmp(y)
    dmd_tmp(x) = dmd_tmp(x) - sup_tmp(y)
    sup_tmp(y) = -2
    For z = 1 To dmd_cnt
        ibfs(y, z) = -2
    Next z

    If QS = True Then
        hconnect_line(y).Visible = True
    End If
Else
    If dmd_tmp(x) > sup_tmp(y) Then
        bfs(y, x) = sup_tmp(y)
        var_lbl((y - 1) * (dmd_cnt) + x) = bfs(y, x)
        sup_lbl(y) = "X"
        dmd_lbl(x) = dmd_tmp(x) - sup_tmp(y)
        dmd_tmp(x) = dmd_tmp(x) - sup_tmp(y)
        sup_tmp(y) = -2
        For z = 1 To dmd_cnt
            ibfs(y, z) = -2
        Next z

        If QS = True Then
            hconnect_line(y).Visible = True
        End If
    Else
        bfs(y, x) = dmd_tmp(x)
        var_lbl((y - 1) * (dmd_cnt) + x) = bfs(y, x)
        dmd_lbl(x) = "X"
        sup_lbl(y) = sup_tmp(y) - dmd_tmp(x)
        sup_tmp(y) = sup_tmp(y) - dmd_tmp(x)
        dmd_tmp(x) = -2
        For z = 1 To sup_cnt

    End If
End If
The following code is to check for optimality of the current solution. This function starts by creating the ‘u’ and ‘v’ variables associated with each row and column. These values are discovered using a while loop. Starting in the top left cell and making the assumption that $v_1 = 0$, the function searches left to right row by row looks for a value that connects the first v with the first u. Then the loop keeps searching left to right and row by row until the end of the matrix.

If all the $u$ and $v$ values are not filled out then the loop starts over at the beginning, the top left cell looking left to right and row by row until all $u$’s and $v$’s are found. When all $u$’s and $v$’s are found the function calculates the adjusted cost matrix. When the adjusted cost matrix is complete the function checks the adjusted cost matrix looking for positive values, if one is found the solution is suboptimal and that message is displayed. This function places the output information on a form called form two. Form two has MSflexgrids to display all of its information, but there is no important code in form two.

Private Function optimal()
Dim u() As Long
Dim v() As Long
Dim ut() As String
Dim vt() As String
Dim z_tot As Long
Dim x, y, z As Long
Dim stopp As Boolean
Dim a, b As Long
ReDim u(sup_cnt)
ReDim v(dmd_cnt)
ReDim ut(sup_cnt)
ReDim vt(dmd_cnt)
ReDim cm_bar(sup_cnt, dmd_cnt)
For a = 1 To sup_cnt

ibfs(z, x) = -2
Next z

If QS = True Then
  vconnect_line(x).Visible = True
End If
End If
End If
End Function
u(a) = 0
ut(a) = "O"
Next a

For a = 1 To dmd_cnt
  v(a) = 0
  vt(a) = "O"
Next a

Form2.Label2.Caption = ""
Form2.Label5.Caption = ""
Form2.Label6.Caption = ""
Form2.MSFlexGrid1.Clear
Form2.MSFlexGrid2.Clear
Form2.MSFlexGrid3.Clear

Form2.MSFlexGrid1.TextMatrix(0, 0) = "U"
Form2.MSFlexGrid2.TextMatrix(0, 0) = "V"

u(1) = 0
Form2.MSFlexGrid1.TextMatrix(1, 0) = 0
ut(1) = "F"

Do While stopp = False
  For a = 1 To sup_cnt
    For b = 1 To dmd_cnt
      If vt(b) = "O" And ut(a) = "F" And bfs(a, b) >= 0 Then
        v(b) = cost_matrix(a, b) - u(a)
        Form2.MSFlexGrid2.TextMatrix(b, 0) = v(b)
        vt(b) = "F"
      End If
      If vt(b) = "F" And ut(a) = "O" And bfs(a, b) >= 0 Then
        u(a) = cost_matrix(a, b) - v(b)
        Form2.MSFlexGrid1.TextMatrix(a, 0) = u(a)
        ut(a) = "F"
      End If
    Next b
  Next a
  stopp = True
  For a = 1 To sup_cnt
    If ut(a) = "O" Then
      stopp = False
    End If
  Next a
  For a = 1 To dmd_cnt
    If vt(a) = "O" Then
      stopp = False
    End If
  Next a
Loop
stopp = True
For a = 1 To sup_cnt
    For b = 1 To dmd_cnt
        If bfs(a, b) > -1 Then
            z_tot = z_tot + bfs(a, b) * cost_matrix(a, b)
            If Form2.Label5.Caption = "" Then
                Form2.Label5.Caption = "(" + CStr(cost_matrix(a, b)) + " * X" + CStr(a) + CStr(b) + ")"
            Else
                Form2.Label5.Caption = Form2.Label5.Caption + " + " + "(" + CStr(cost_matrix(a, b)) + " * X" + CStr(a) + CStr(b) + ")"
            End If
        End If
        Form2.MSFlexGrid3.TextMatrix(a - 1, b - 1) = u(a) + v(b) - cost_matrix(a, b)
        If u(a) + v(b) - cost_matrix(a, b) > 0 Then
            stopp = False
        End If
    Next b
Next a

Form2.Label6.Caption = z_tot
If stopp = True Then
    Form2.Label2.Caption = "Optimal"
Else
    Form2.Label2.Caption = "Not Optimal"
End If

End Function