

**investigation into the optimal transportation model**

by jennifer m

Table of Contents

[Analysis 3](#_Toc33377744)

[Background: 3](#_Toc33377745)

[Identification of the problem: 3](#_Toc33377746)

[Project outline: 3](#_Toc33377747)

[Current System: 4](#_Toc33377748)

[Research: 4](#_Toc33377749)

[Assumptions Made: 5](#_Toc33377750)

[Prospective Users: 5](#_Toc33377751)

[User Interviews: 5](#_Toc33377752)

[User Requirements: 6](#_Toc33377753)

[Objectives: 7](#_Toc33377754)

[Limitations with the Tkinter library: 8](#_Toc33377755)

[Acceptable limitations of the transportation model: 8](#_Toc33377756)

[Data Flow Diagram: 8](#_Toc33377757)

[Design 9](#_Toc33377758)

[Overview: 9](#_Toc33377759)

[Structure of the Program: 9](#_Toc33377760)

[Classes: 10](#_Toc33377761)

[Data Structures: 10](#_Toc33377762)

[Two-Dimensional Array: 10](#_Toc33377763)

[Adjacency Matrix: 11](#_Toc33377764)

[Lists: 11](#_Toc33377765)

[Data Validation: 11](#_Toc33377766)

[Interface Design: 12](#_Toc33377767)

[The Second Window: 13](#_Toc33377768)

[The Third Window: 14](#_Toc33377769)

[North West Corner Method: 15](#_Toc33377770)

[Vogel’s approximation method: 17](#_Toc33377771)

[The Least Cost method: 20](#_Toc33377772)

[The Stepping Stone Assessment: 22](#_Toc33377773)

[Techniques Used: 28](#_Toc33377774)

[Libraries: 28](#_Toc33377775)

[Log of the stages of the investigation: 28](#_Toc33377776)

[Technical Solution 29](#_Toc33377777)

[Classes Created: 29](#_Toc33377778)

[Libraries used: 34](#_Toc33377779)

[Main Program: 35](#_Toc33377780)

[Testing 65](#_Toc33377781)

[First Window: 65](#_Toc33377782)

[Second Window: 65](#_Toc33377783)

[Third Window: 65](#_Toc33377784)

[Test evidence 66](#_Toc33377785)

[Test scenarios 66](#_Toc33377786)

[Evaluation 77](#_Toc33377787)

[Comparison of the project against the objectives: 77](#_Toc33377788)

[User feedback: 78](#_Toc33377789)

[Possible improvements: 78](#_Toc33377790)

[Outcome of the investigation 78](#_Toc33377791)

# Analysis

## Background:

Companies worldwide are ever exploring new ways in order to find the most resource-efficient, straightforward methods for transporting their merchandise to warehouses. These methods should produce the lowest costs for the corporation.

Many manufacturing companies employ the use of transportation management due to the quick implementation and cost-effective methods. This can lead to lower prices for both the business and the customer, reducing the amount of local competition a company has.

## Identification of the problem:

During the transportation process, different amounts of labour are required depending on the time of year; this can include drivers and factory workers. In addition, driver shortages are common; consequently, space in lorries must not be wasted. As well as this, there is a limited amount of space where merchandise can physically be stored before dispatching to stores. Therefore, this factor must also be taken into account when calculating the optimal transportation system.

Due to the rising awareness of the effect of climate change, there are tighter regulations about the level of emissions a company can produce. If the company creates more emissions than the regulated amount, the company will have to pay taxes, decreasing annual profits. As well as this, there is a fluctuation in pricing for fuel, which cannot be predicted, so careful planning must be made to factor in these additional costs and ensure maximum profits are achieved.

Logistics companies currently have the most up-to-date systems for transportation, but emerging markets and the world’s supply networks continue to develop, so these companies will need to fine-tune or even transform their systems to cope with a change in supply and demand and to outcompete competitors. In order to find the cost-effective systems, numerous calculations must be carried out, more often than not by hand, which can take up too much time and are error prone.

Overall, business models rely on the data provided, they cannot anticipate the future, meaning if there was a sudden change in one of the factors, the entire business model would need to be recalculated. This can take long time periods that a company does not necessarily have. Therefore, by creating a program that does this for you, companies do not have to spend as much time creating a solution.

## Project outline:

I am going to explore the problems related to proficient transportation by creating the optimal model for transporting goods from company-owned factories to their warehouses. This program could potentially be used by companies and businesses that are trying to maximise their profits.

## Current System:

There are multiple websites explaining what the transportation problem is and they contain detailed explanations of the different methods that can be used to solve it. Also, there are websites that allow a user to enter data and the website will then produce a table of allocations using a selected method.

Advantages of the current system:

* The user can find a solution to their transportation problem

Disadvantages:

* Any results found cannot be easily compared and no conclusions are described to the user.
* The results given are not easily understood, the table is not visually accommodating

These methods are complex and may not be understood by the typical worker. Therefore, by creating this program, the user can visually appreciate the idea of how the methods work, inspiring reassurance that they are using the most efficient method. They will also find the most profitable solution quickly. This will help companies embrace digitalisation, where they were previously reluctant to adopt it.

## Research:

Transportation companies invest around 5% of their annual revenues into digital operations, including the increased use of technology to unlock new business models and optimise profits[[1]](#footnote-1).This program will use some of the most popular methods of optimising profits.

The number of goods transported has increased by 4.5% during 2016 – 2017 for the whole of the EU. The United Kingdom alone increased their road freight by 13%, from 122 billion T/km to 174 billion T/km [[2]](#footnote-2). This shows the ever-increasing demand for transportation models within businesses worldwide.

After researching the numerous ways of calculating an initial feasible solution, I found the three most common methods were the North West Corner method, Vogel’s approximation method and the Least Cost method. They all involve using a matrix and the costs of each individual journey from a factory to a warehouse. The mathematical calculations involved are described later on.

These methods rely on the demand being equal to the supply, if this is not true a dummy source will be added to the matrix. The demand/ supply of the source is equal to the difference between the total supply and demand. If there is more supply than demand, a dummy column will be added to and a row for the opposite scenario. The costs of the row/ column are equal to zero, so the overall cost is not affected.

As well as this, I discovered an algorithm that optimises the initial feasible solution, it is called the Stepping Stone Method. There are two different scenarios when the algorithm can be used: when a solution is degenerate or when a solution is non-degenerate.

A solution is said to be degenerate if the number of rows (m) summed with the number of columns (n) subtract 1 is not equal to the number of locations where allocations have bene made ie, m + n –1 ≠ number of allocation locations.

The conditions of the Stepping Stone method vary depending upon which scenario the data is classed as.

## Assumptions Made:

Before implementing the program, several assumptions have to be made. The assumptions are as follows:

* Products are homogenous, meaning the product has the same features no matter which supplier/factory it comes from.
* Shipping costs per unit are the same no matter the quantity.
* One route is chosen between origin and destination, so the cost isn’t affected by the real-life journey taken.
* All journeys are taken using the same form of transport, so there is no variance in cost.

## Prospective Users:

Student 1 is a business student who wants to find the cheapest way their online shopping can be transported from warehouses to their local delivery centres.

Student 2 is an economics student who hopes to become an entrepreneur and, by using this software, hopes to gain a better understanding of the processes involved with transporting goods.

## User Interviews:

I gave an outline of my project before asking the following questions.

**Outline:**

‘Products are to be transported from a number of factories to a number of warehouses at the minimum possible cost. Each factory is supplied a fixed number of units of the products and each destination has a fixed demand.’

**Questions:**

1. Why is it important for a business to maximise profits?
2. What risks are involved within the transportation of goods?
3. Can a company benefit from the way their products are transported?
4. What should the aims/ objectives of the investigation be?
5. Do you think there will be any limitations?
6. What important features should the software have?

**Student 1:**

1. It is important to maximise profits so a business has enough cash flow for the following reasons: to survive, to expand and to gain more market share. This will drown out other competitors. As well as, keep investors interested in your business and attract new investors.
2. The main risks are the physical travelling conditions ruining goods or preventing journeys from being taken. Also, if products are damaged, the insurance is hard to determine as it is hard to say who is at fault.
3. A company can benefit if they carry out the transportation through their own company as it will be cheaper for them. Whereas, if they use external resources it costs a lot more as many extra charges are involved.
4. The aims should be to find a cost of a business using their own resources to transport goods given certain supply and demand constraints. As well as, finding the most cost-efficient way of doing this.
5. Limitations of your software may be that it can account for the information it is given, but can’t take into account external factors, such as, weather and change in per unit cost.
6. The software should have an interface that is easily understood by a user who only has a vague idea of what the transportation model is. It should also allow the user to choose which method they want to use to find a solution.

**Student 2:**

1. It is important to maximise profits because the main objective for a company is to make a profit of their goods and maximising returns for their shareholders.
2. The time scale of the transportation can affect products if they are perishable, as it can lead to a big loss of revenue because products are no longer consumable. Also, there is a risk of loss or theft of the goods, which would reduce profit.
3. A company can benefit if transport is cheaper. If transportation costs are low, marginal cost goes down so profit per unit is higher**.1** Also, if a company has a transport system where cost, time and reliability are taken into account, they can produce a good economy of sale**2** and allow the company to develop.
4. The main aim should be to find the cheapest and most efficient transport system available for a business.
5. The main limitation will be the time limit you have to write the code.
6. The software should produce a graph showing the final solution, including which factories are delivering to which warehouses.

## User Requirements:

1. The program must have a suitable interface that is easy to use and understand.
2. The program must have a section that allows the user to input their data into an adjacency matrix.
3. There must be an option where any of the three algorithms can be selected by the user.
4. The results of a method must be shown clearly and in a way that can be understood by the user. A graphical solution should be used.
5. The user must be able to access explanations of the information being shown if they are confused.

1 – marginal costs are the additional costs for each unit sold.

**2** – economy of sale is when there is a proportionate saving in cost gained by an increased level of production.

## Objectives:

1. Create a Tkinter interface
2. There must be a menu, entry page, results page(s)
3. Menu must have:
   1. An image of a lorry
   2. Input boxes for user’s data:
      1. Number of factories
      2. Number of warehouses
      3. Total supply
      4. Total demand
   3. An execution button that takes the user to the entry page
      1. When clicked on, the data in the entry boxes are collected and used to create the adjacency table for the next page
   4. An explanation of how to use the program:
      1. Shown from the click of a ‘help’ button
      2. An information box will pop up explain the corresponding window
4. Entry page must:
   1. Display a simple adjacency matrix with input boxes
      1. There should be labels for the row and columns showing what each row/ column is for
   2. Have a drop-down menu for methods
   3. Have a help menu that gives information about the algorithms and the adjacency matrix
5. Results page must:
   1. Display a table showing quantities assigned for each route
   2. Display a network graph of routes taken
   3. The original solution should also be shown with the same features if requested by the user
6. There must be working buttons and drop-down menus
   1. Use the built-in widgets to create them
   2. Use meaningful identifiers to show what each button/ menu is for
   3. Each button/ menu option must call a function to produce a result
      1. Help buttons will provide accommodating information
      2. The Menu button will call the three different methods
      3. The execution button will use the data entered to create an adjacency matrix
7. Calculate initial costs for every selected method
   1. Data will be collected from the user before the selected algorithm is executed
   2. The selected algorithm will be executed on the collected data
   3. The solution is checked to see if it is degenerate
      1. Using the ‘m+n-1=allocations made’ formula
8. Optimise initial solution(s)
   1. Use the Stepping Stone algorithm for optimality
   2. Displays the results on Tkinter interface

## Limitations with the Tkinter library:

When using the Tkinter library there are some limitations, including the following:

* The Canvas library cannot be used on the same window as the built-in Tkinter widgets, such as, Label and Button. This means more than one identifier must be used to refer to the windows.
* Code can be hard to debug since Tkinter widgets are not python objects; Tkinter provides a package around the Tk widgets which means you can get unusual error messages

## Acceptable limitations of the transportation model:

For this investigation, there will be a few limitations because of the time scale of the project. However, they have all been agreed with the prospective users.

* The network graph will be abstracted so the real-life locations of factories and warehouses will not necessarily be the same as the positioning of the nodes on the graph.
* Degenerate solutions will not be optimised as the algorithm requires different conditions for degenerate solutions compared to non-degenerate solutions.
* External factors cannot be taken into account, including the conditions for travel (weather).

## Data Flow Diagram:

USER INPUT:

No. factories/ warehouses

USER INPUT:

Total supply / demand

Store user data

as variables

Create entry matrix

using user data

Create first window

Create second window

Methods can be clicked as many times as wanted

USER INPUT:

Costs and demands/

supply into matrix

Create third window

Network graph / allocation table created

USER SELECT:

method

# Design

## Overview:

The program can be thought of as two parts: the first is finding the initial feasible solution of a matrix, and the second optimising this solution. The results from both parts should be shown to the user through a Tkinter interface.

The user interface will comprise of a table where the user can enter data, such as, costings and number of factories. The interface should also have different input boxes where the user can select which method they would like to use.

## Structure of the Program:

The following diagram shows the steps of the overall system.

Step 1

Input Matrix Interface

Balanced Problem

Unbalanced Problem

Dummy Source

North West Corner Method

Least Cost Method

Vogel’s Approximation Method

Menu Interface

Stepping Stone Optimisation

Results Page

Step 2

Step 3

Step 4

Step 5

Step 6

## Classes:

Two classes will be used to aid the design of the interface. The classes have been used as they are easy to maintain and make errors easier to debug. Also, by using a class, the behaviours of the data are predefined, so the methods can be used in any context. In this context as a new instance of a classes is created every time a class is called, the results shown on the interfaces can differ depending upon the user’s inputs. The classes and their methods are shown below.

|  |
| --- |
| tableClass |
| -\_\_init\_\_(self, rows, columns, factories, warehouses, totalSupply, totalDemand, matrix, entries)  +createTable(self) |

|  |
| --- |
| analysisClass |
| -\_\_init\_\_(self, cost, rows, columns, factories, warehouses, totalSupply, totalDemand, method, allocationArray, blankCanvas, degenerate)  -\_\_calculateTotal(self)  -\_\_balancedChecked(self)  -\_\_labelAllocationArray(self)  +analysisPage(self)  +networkGraph(self)  +resultsTable(self) |

The ‘tableClass’ class should produce the input matrix to be used on the interface. It should label all rows and columns and show the total supply and demand.

The ‘analysisClass’ class should create a network graph (‘networkGraph’ method) and create a table of allocations to be shown to the user on the results page (‘resultsTable’ method). The table of allocations should have each row and column labelled (‘\_\_labelAllocationArray’ method) and the total allocations of each row and column should be shown (‘calculateTotal’ method). Information describing the graph, table and the final cost should also be displayed (‘analysisPage’ method). The ‘\_\_balanceChecked’ method will ensure any dummy sources are not included in the results.

The ­\_\_init\_\_ method in both classes initialises any variables that need to be used within the methods.

## Data Structures:

### Two-Dimensional Array:

The data entered by the user into the input matrix is stored as a 2D array. This is so the three main algorithms can work on each cell individually and in relation to other cells. It is the most efficient way of storing the data as other ways will be harder to manipulate and maintain. An example would be storing each row as separate arrays, which would make the data practically impossible to manipulate.

### Adjacency Matrix:

Once the initial feasible solution has been found, the data is stored as an adjacency matrix, where one cell represents the units being transported from a factory to a warehouse. This allows the initial solution to be optimised. It also shows the connections between factories and warehouses.

### Lists:

Lists have more flexibility than arrays, for example they can store a variety of data. Also, the length of the list does not need to be known in order to be created. A list is used to store the vertices of each path created in the Stepping Stone Algorithm. This is so incorrect vertices can easily be removed using the built-in function .pop().

## Data Validation:

The code is based heavily upon the data entered by the user. To prevent the program from crashing, all user inputs will be validated. The following validation checks have been implemented:

* For the first window, the data collected will be used in a try/ catch to check it is an integer, if not an appropriate error will be shown. The data entered must be whole numbers above one, so a matrix can be created and the is valid for the three main algorithms. This means I will have to produce error messages for any non-integer positive characters, including letters, decimals and negative numbers.
* A try/catch will also be used for the data entered in the input matrix, on the second window. This is the easiest way to validate the data, as it must first be collected from an array of objects and then stored in a matrix. Further validation is then carried out on the input data using selection statements and appropriate errors are shown if the data is invalid.
* The costs data entered in the entry matrix must be whole numbers above zero, so letters and negative numbers are not accepted. There are also certain circumstances when the algorithms will not be executed, such as, when all costs are zero there is no solution as all possible solutions will have a zero cost. Also, if all the costs are the same, then all solutions will amount to the same cost so this scenario is not accepted.
* The demand row and supply column must be positive integers that add up to the total supply and total demand values previously entered by the user. This is an integrity check as it checks the user has entered the correct supplies and demands.

The names of the factories and warehouses can be edited and any character can be entered. However, there is no validation for this data because it is not collected or used in any part of the program. It is editable so the user can change the pre-set values to their factory and warehouse names, so it is easier for the user to understand what each cell represents.

All validation is checked in the testing section.

## Interface Design:

The interface is designed to hide the complexity of the algorithms that are being used. I will be using Python 3 to write my code and I will be importing the Tkinter library to create the interface. The user should be accustomed to the widgets used on the interface as they are designed to look similar to the ones used on the operating system the program is being run on. This is through the use of the Themed Tkinter widgets (ttk).

The First Window:

The Transportation Model

No. Factories

No. Warehouses

Total Supply

Total Demand

Execution

Help

1.

2.

5.

3.

4.

* + 1. An image of a lorry will be added here as a visual aid of the transportation problem to the user.
    2. These are entry boxes, where the user can enter numeric data in to be collected and used to create the next window.
    3. These are labels used to describe what each entry box is for. This is to help the user understand which data to enter where.
    4. This is a button used to display the next window to the user.
    5. This is a button used to display information about the current window to the user. This is to ensure the user fully understands what each label represents.

## The Second Window:

Factory |

Warehouse

Supply

Total:

Demand

Methods Menu

Help Menu

The Transportation Model

W1

F1

F2

F3

W2

W3

3.

1.

2.

4.

5.

6.

7.

This is a table that will be used to find solutions to the user’s problem.

These are labels stating what the row/ column represents.

These are entry boxes, where the user can enter the names of their factories and warehouses. They have been initially labelled as a factory/ warehouse number, so the user does not need to enter data into these boxes. This can speed up the time the user takes to find a solution, making the interface more user friendly.

These are entry boxes for the cost of travelling from the factory for that row, to the warehouse for that column.

These are entry boxes for the supply/ demand for the factory/ warehouses on that row/ column.

This is a drop-down menu showing the three different methods that can be used to find an initial solution. The user can only click one at a time, but the menu can be used as many times as needed.

This is another drop-down menu that gives the user a choice of information. This includes what data needs to be entered and where in the table. As well as, an explanation of what the methods are.

This shows an adjacency matrix.

These are the units to be transported from the

Specified factories to the specified warehouses.

F1

F2

F3

Method Name

W1

W2

W3

F|W

W1

W2

W3

F1

F2

F3

Total

Total

A network graph of the different routes taken.

The supplies to be transported are shown.

The total cost for this method is:

1.

2.

3.

4.

7.

6.

5.

## The Third Window:

1. The name of the method is used as the window title so the user does not get confused if they find results for multiple methods.
2. This is text information shown to the user to help aid understanding of what the graph and table are.
3. These are the nodes of the network graph that represent each factory and warehouse.
4. This is where any connections (allocations) are shown from a factory to the corresponding warehouse.
5. This is an adjacency matrix, where the final allocations are shown.
6. The number of units for each exchange between a factory and warehouse will be shown in these boxes.
7. The total allocations bring sent/ received at a factory/ warehouse are shown in these cells.

## North West Corner Method:

To carry out this method you start in the upper left corner of an empty matrix and assign as much of the supply as you can to saturate the demand. However, you must stay within the supply and demand constraints. If the supply is in surplus, you note down the remaining supply in the supply box and set the demand as zero. Then you move horizontally to the right and do the same again, until the supply of that factory is used up.

Once the supply of a factory is used up, move down a row to the next factory and repeat the process. Once you have done this for all factories, the supply and demand constraints will be fully saturated and you can calculate the total cost. Do this by multiplying the units assigned to each warehouse by the transportation cost given by the user.

I wrote the pseudocode for each method before writing the actual code. This is to make the code writing easier and quicker because it helped to clarify my thinking. The pseudocode for this method is as follows:

#allocationArray is an array of zeros and data is predefined from the user’s inputs

y 🡪 0

x 🡪 0

allocations 🡪 0

supply 🡪 data[x][-1]

demand 🡪 data[-1][y]

WHILE allocations != totalSupply and allocations != totalDemand

IF demand == supply THEN

allocationArray[x][y] = demand

allocations += supply

y += 1

x += 1

supply 🡪 data[x][-1]

demand 🡪 data[-1][y]

ELSE IF demand < supply THEN

allocationArray[x][y] = demand

allocations += demand

supply -= demand

y += 1

demand 🡪 data[-1][y]

ELSE THEN

allocationArray[x][y] = supply

allocations += supply

demand -= supply

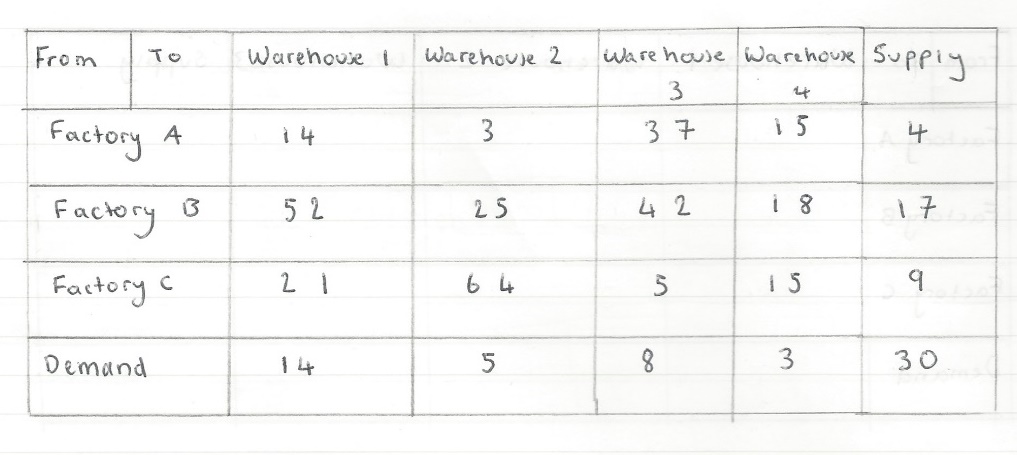
x += 1

supply = data[x][-1]

END IF

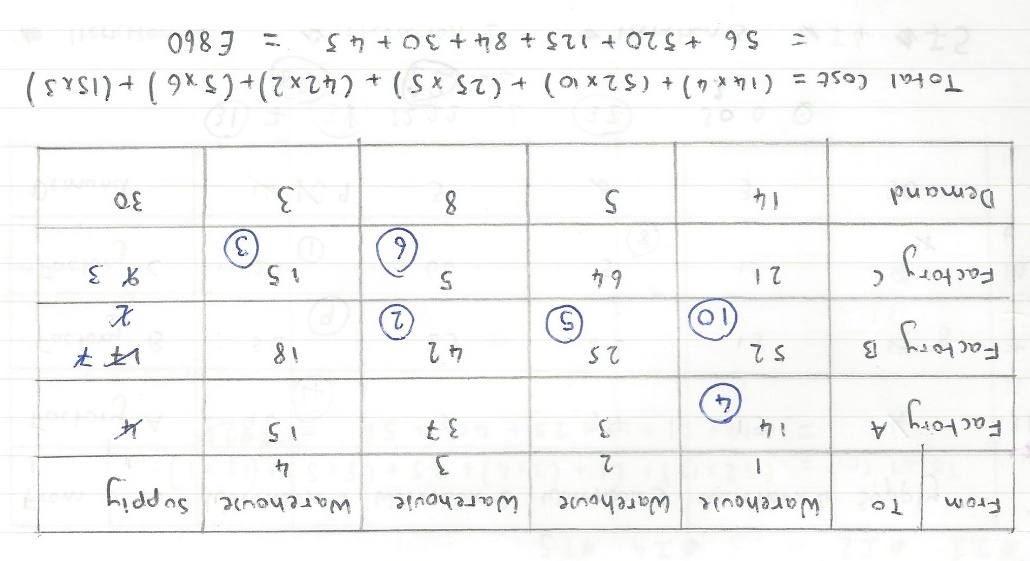
END WHILE

I carried out dry runs on a random matrix so I could further understand the way I which the method works. This was the data matrix upon which I worked the method:

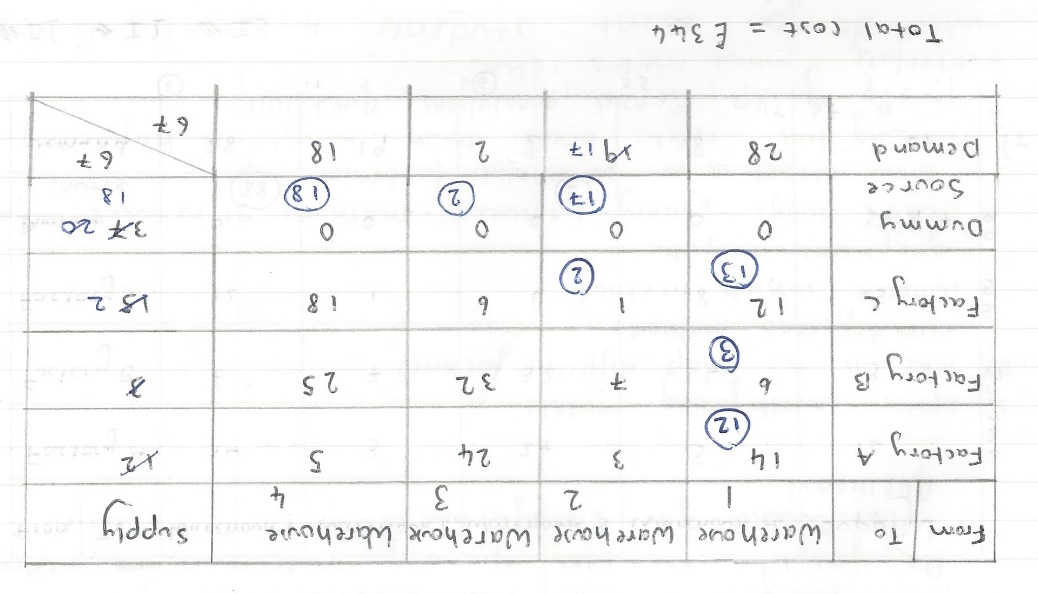


I tried a balanced solution (total supply = total demand) and an unbalanced solution (total supply != total demand).

The dry run of the balanced solution:



The dry run of the unbalanced solution:



## Vogel’s approximation method:

To find the solution to this method first calculate the difference between the two cells with the lowest cost for each row and column, where allocations have not yet been made. Next, select the row/ column with the largest difference.

#rows, columns, factories, warehouses, totalSupply, totalDemand and data are all predefined from user inputs

#allocationArray is an array of zeros

allocations 🡪 0

WHILE allocations != totalSupply AND allocations != totalDemand

FUNCTION find difference

#a function that finds the difference between the two cells with the lowest cost for each row/ column is called

FUNCTION highest row difference

RETURN xIndex

#a function that compares all row differences and returns the x coordinate of the row with the highest difference

FUNCTION highest column difference

RETURN yIndex

#a function that compares all column differences and returns the y coordinate of the row with the highest difference

lowestCost 🡪 inf

IF yIndex == None THEN

FOR position IN RANGE 0 🡪 warehouses

IF data[xIndex][position] < lowestCost THEN

lowestCost 🡪 data[xIndex][position]

yIndex = position

ELSE THEN

Pass

END IF

END FOR

ELSE THEN

FOR position IN RANGE 0 🡪 factories

IF data[position][yIndex] < lowestCost THEN

lowestCost 🡪 data[position][yIndex]

xIndex 🡪 position

ELSE THEN

Pass

END IF

END FOR

END IF

Then, assign the largest quantity within the supply and demand constraint in the cell with the lowest cost. In the case of a tie choose the row/ column where you can assign the largest quantity to the cell with the lowest cost. After, ignore the row/column that has been saturated for the rest of the calculations. Repeat until all the demand and supply requirements are met. Once saturated, you can calculate the total cost by multiplying the unit assigned to each cell with the cost concerned.

supply 🡪 data[xIndex][-2]

demand 🡪 data[-2][yIndex]

IF supply == demand THEN

allocationArray[xIndex][yIndex] 🡪 supply

allocations += supply

supply 🡪 0

demand 🡪 0

FOR r IN RANGE 0 🡪 rows

data[r][yIndex] 🡪 inf

END FOR

FOR c IN RANGE 0 🡪 columns

data[xIndex][c] 🡪 inf

END FOR

data[xIndex][-2] 🡪 0

data[-2][yIndex] 🡪 0

ELSE IF demand < supply THEN

allocationArray[xIndex][yIndex] 🡪 demand

allocations += demand

supply -= demand

demand 🡪 0

FOR r IN RANGE 0 🡪 rows

data[r][yIndex] 🡪 inf

END FOR

data[xIndex][-2] 🡪 supply

data[-2][yIndex] 🡪 0

ELSE THEN

allocationArray[xIndex][yIndex] 🡪 supply

allocations += supply

demand -= supply

supply 🡪 0

FOR c IN RANGE 0 🡪 columns

data[xIndex][c] 🡪 inf

END FOR

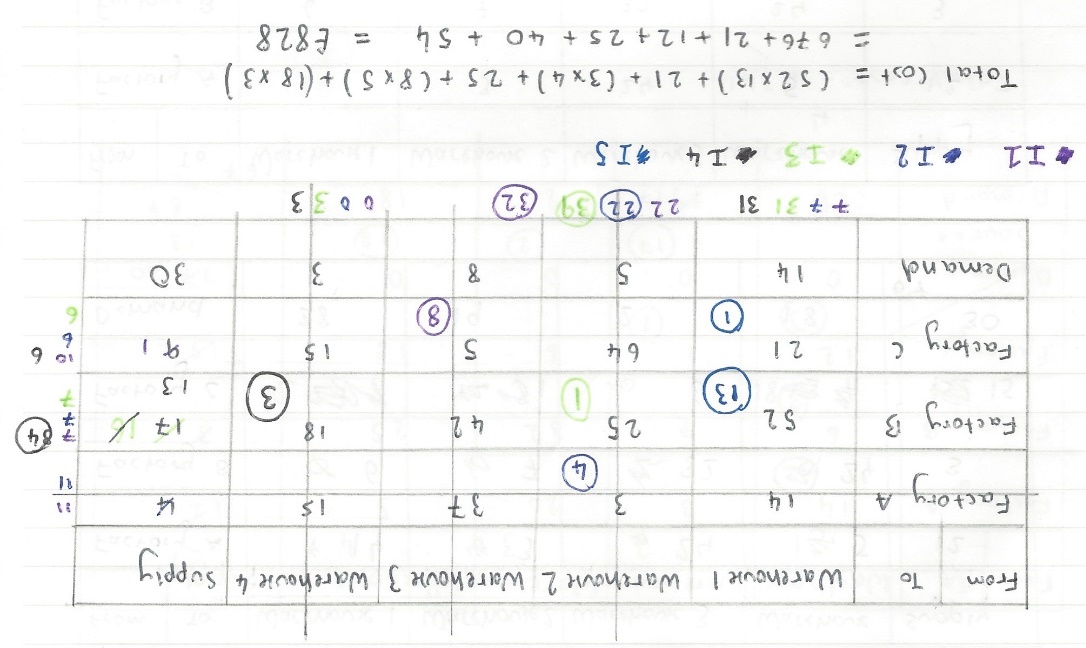
data[xIndex][-2] 🡪 0

data[-2][yIndex] 🡪 demand

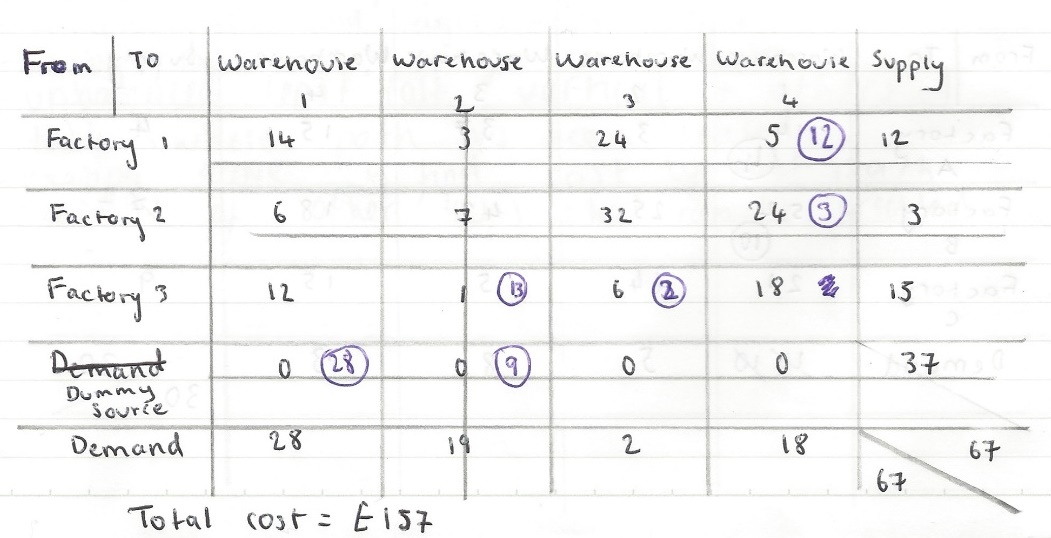
END IF

END WHILE

The balanced dry run I carried out for this method:



The unbalanced dry run for this method:



## The Least Cost method:

In order to carry out this method, first you find the cell with the lowest cost in the cost matrix.

#rows, columns, totalSupply, totalDemand and data are all predefined from user inputs

#allocationArray is an array of zeros

allocations 🡪 0

IF totalDemand < totalSupply THEN

totalAllocationsNeeded 🡪 totalSupply

ELSE THEN

totalAllocationsNeeded 🡪 totalDemand

END IF

WHILE allocations < totalAllocationsNeeded

value 🡪 inf

FOR x IN RANGE 0, rows

FOR y IN RANGE 0, columns

IF data[x][y] < value THEN

xIndex 🡪 x

yIndex 🡪 y

value = data[x][y]

ELSE IF data[x][y] == value THEN

IF data[x][y] != inf THEN

currentSupply 🡪 data[xIndex][-1]

currentDemand 🡪 data[-1][yIndex]

newSupply 🡪 data[x][-1]

newDemand = data[-1][y]

IF newSupply > currentSupply THEN

xIndex 🡪 x

yIndex 🡪 y

ELSE IF newDemand > currentDemand THEN

xIndex 🡪 x

yIndex 🡪 y

ELSE THEN  
 Pass

END IF

ELSE THEN

Pass

END IF

ELSE THEN

Pass

END IF

Then allocate the maximum supply in order to comply with the demands. If there is a tie, the cost where maximum supply can be assigned should be chosen. Repeat until all supply and demand are saturated. If there is a minimum cost where the supply has already been used, move onto the next lowest cost. Once saturated, the total cost can be calculated by multiplying the assigned quantity to each cell by the cost of that cell.

supply 🡪 data[xIndex][columns]

demand 🡪 data[-1][yIndex]

IF supply == demand THEN

allocationArray[xIndex][yIndex] 🡪 demand

allocations +=demand

FOR r IN RANGE 0 🡪 rows

data[r][yIndex] 🡪 inf

END FOR

FOR c IN RANGE 0 🡪 columns

data[xIndex][c] 🡪 inf

END FOR

data[xIndex][-1] 🡪 0

data[-1][yIndex] 🡪 0

ELSE IF demand < supply THEN

allocationArray[xIndex][yIndex] 🡪 demand

allocations += demand

supply -= demand

FOR r IN RANGE 0 🡪 rows

data[r][yIndex] 🡪 inf

END FOR

data[xIndex][-1] 🡪 supply

data[-1][yIndex] 🡪 0

ELSE THEN

allocationArray[xIndex][yIndex] 🡪 supply

allocations += supply

demand -= supply

FOR c IN RANGE 0 🡪 columns

data[xIndex][c] 🡪 inf

END FOR

data[xIndex][-1] 🡪 0

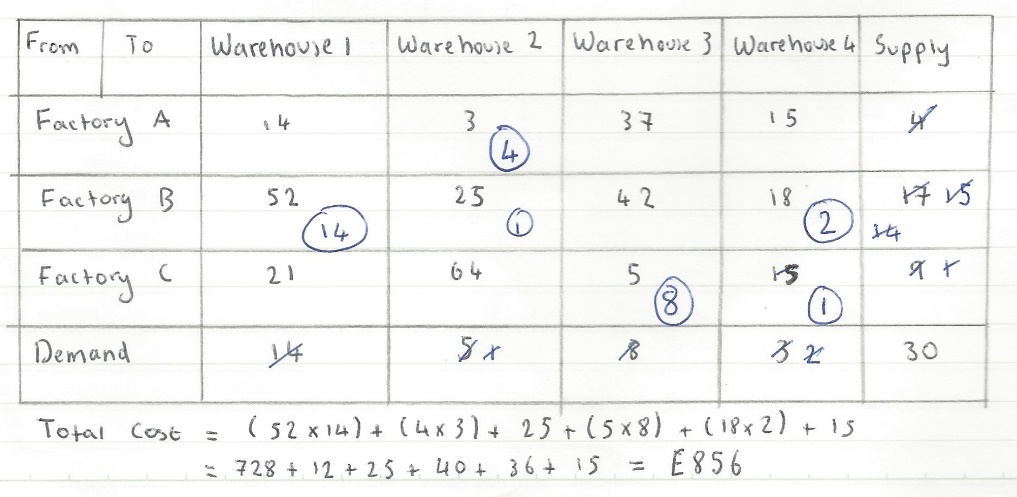
data[-1][yIndex] 🡪 demand

END IF

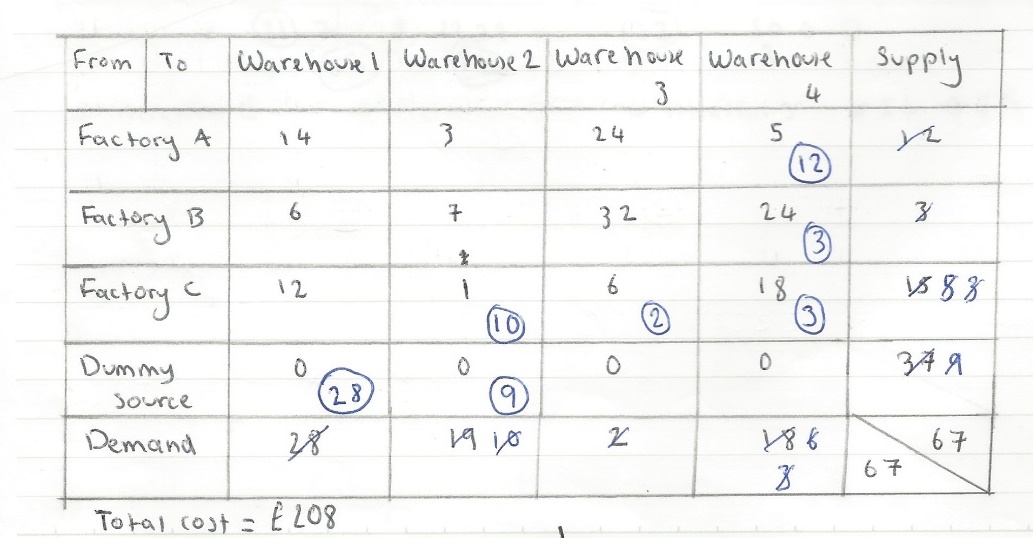
END WHILE

The dry runs for this algorithm are as follows:

Balanced:



Unbalanced:



These dry runs were very time enduring, which supports the idea that my program makes finding solutions to the transportation problem much faster and also reduces the chance of error.

## The Stepping Stone Assessment:

Before using this method, you must check the initial feasible solution is not degenerate.

#factories and warehouses are determined from the user input

IF allocations < (factories+warehouses-1) THEN

RETURN True

ELSE THEN

RETURN False

If the solution is not degenerate, a path must be created for each empty cell in the allocation matrix. In order to create this path, the following rules should be applied:

* The first and last vertex of the path is the empty cell
* Only one cell in the loop should be unoccupied (have no allocations)
* Only horizontal and vertical movement is allowed

Once the loop has been created, starting with the empty cell, assign ‘+’ and ‘-’ alternatively to the cost of each vertex of the path. The sum of the new integers is the cost of the path. If the new cost of that path is greater than or equal to zero, then no changes are needed as it is the optimum path. If the cost is negative, there is a way to reduce the overall cost. To reduce the cost, first find the smallest allocation made out of the negative vertices. Then subtract this quantity from the vertices with a ‘-‘ sign and add to the vertices with a ‘+’ sign. This balances the demand and supply requirements. Repeat this process for all empty cells. Then repeat the entire process until all path costs are greater than or equal to zero.

This is a complex algorithm as there are many conditions that must be followed in order for the path can be created. Therefore, I have written the pseudocode before trying to implement the actual code. There are comments throughout the pseudocode explaining in detail all conditions and steps taken to achieve optimisation.

#allocationArray stores the initial feasible solution stored

optimalSolution 🡪 False

The following section of code is repeated until the final solution can no longer be made cheaper.

currentNetCost 🡪 0

WHILE optimalSolution == False

FUNCTION empty cells

RETURN skipVertex

#a function that creates an array of cells where no allocations have been made

FOR position IN RANGE 0 🡪 LEN(emptyCells)

vertices 🡪 []

x 🡪 emptyCells[position][0]

A path is created for each empty cell in the allocationArray.

y 🡪 emptyCells[position][1]

vertices.APPEND([x,y])

FUNCTION column search

RETURN vertices

#a function that finds the path of an empty cell and returns the vertices of the path in an array

FUNCTION cost change

The optimised cost of the current path is compared with the lowest cost so far.

RETURN netCost

#a function that assigns the cost of the vertices an alternating + and – sign and calculates the new cost of that path

IF netCost < currentNetCost THEN

currentNetCost 🡪 netCost

optimalVertices 🡪 COPY(vertices)

ELSE THEN

Pass

If the optimised cost is the same, no changes will occur since both paths will produce the same output.

END IF

If the net cost of the path is below zero, the path can be optimised.

END FOR

IF currentNetCost < 0 THEN

FUNCTION smallest allocation

RETURN minimumAllocation, newCosts

#a function that finds the smallest allocation made in the current path and stores the new cost for each vertex in an array

FOR j IN RANGE 0 🡪 LEN(optimalVertices)

To optimise the solution, the smallest allocation of the path is added/subtracted to each vertex.

X 🡪 optimalVertices[j][0]

Y 🡪 optimalVertices[j][1]

IF newCosts[j] > 0 THEN

allocationArray[x][y] += minimumAllocation

ELSE THEN

allocationArray[x][y] -= minimumAllocation

END IF

ELSE THEN

Once all path costs are zero or above, the solution is optimised and the algorithm is finished.

optimalSolution 🡪 True

END WHILE

I have also written the pseudocode for the column search function as it can become very complex very quickly. The first section of pseudocode is what is executed when the function has been recursed because no vertex was found.

vertexFound 🡪 False

IF recurse == True THEN

count += 1

If the function is recursed more times than there are warehouses, the two most recent vertices found can not be part of the path.

This is because there are no allocations surrounding them that comply with the conditions required to create a path.

IF count > warehouse THEN

vertexFound 🡪 True

timer += 1

vertices.POP()

vertices.POP()

x 🡪 vertices[-1][0]

ELSE THEN

x 🡪 0

If the function has not been recursed, the x coordinate is set to zero and a new search is started at the top of the current column.

If the function has been recursed less times than there are warehouses, the x and y coordinates are changed to (0, y-coord of the last vertex).

This is so the column search starts again from the top of the column containing the previous vertex.

This means any vertices that have been removed are skipped over and a new vertex can be found.

y 🡪 vertices[-1][1]

END IF

recurse 🡪 False

ELSE THEN

x 🡪 0

END IF

The next section of pseudocode is used to check if an allocation is possible at the current cell.

If the x coordinate is equal to the number of factories, the end of the column has been reached, so this section of code no longer needs to be executed and the function needs to be run again.

WHILE vertexFound == False

index 🡪 [x,y]

IF x == factories THEN

recurse 🡪 True

vertexFound 🡪 True

ELSE IF index IN (skipVertex[i] FOR i IN RANGE(LEN(skipVertex))) THEN

x += 1

ELSE THEN

Any vertices/ coordinates to be skipped are placed in this array. This is to stop them being used as a vertex in the rest of the path.

If the current coordinates are in the array, the x coordinate is increased by one, so that row is missed out on the current iteration.

If a cell has had an allocation made, it is a possibility it can be used as a vertex for the path so it is added to the vertices array.

It is also added to the skipVertex array, so it can’t be used more than once.

IF allocationArray[x][y] >0 THEN

vertices.APPEND([x,y])

skipVertex.APPEND([x,y])

vertexFound 🡪 True

ELSE THEN

Pass

END IF

END IF

END WHILE

The next section of the function pseudocode is executed when a vertex has been found or the condition variables have been set to certain values. This will ensure a specific section of code will be run, so the next vertex can be found.

FUNCTION locations used

RETURN allocations

#a function that calculates the number of locations where allocations have been made and returns this number

FUNCTION degeneracy check

RETURN degenerate

#a function that checks whether a solution is degenerate and returns a Boolean value based on this

If a solution is degenerate, the original solution will be used, so the optimisation algorithm no longer needs to execute.

IF degenerate == True THEN

RETURN None, True

ELSE THEN

FUNCTION path found

RETURN path

#a function that checks if the current path is complete and returns a Boolean condition based on this

FUNCTION vertices check

RETURN vertices

#a function that checks there are no more than two vertices per row/ column and removes any vertices between these vertices

This means a valid path has been created so the function no longer needs to be executed. The path vertices are returned along with a Boolean value that will be used to identify the solution as non-degenerate.

length 🡪 LEN(vertices)

IF path == True AND length > 2 THEN

RETURN vertices, False

ELSE THEN

The row search function is similar to this function, in the way that it finds another vertex on the path. However, instead of searching down a column, it searches across a row. This is so the path has a rectangular sort of shape and not a straight line.

FUNCTION row search

RETURN vertices, y, recurse, count, changeY, timer

FUNCTION path found

RETURN path

IF length >= factories AND path == True THEN

RETURN vertices, False

ELSE THEN

RETURN (RECURSE column search)

#recurses through the current function again to find a vertex

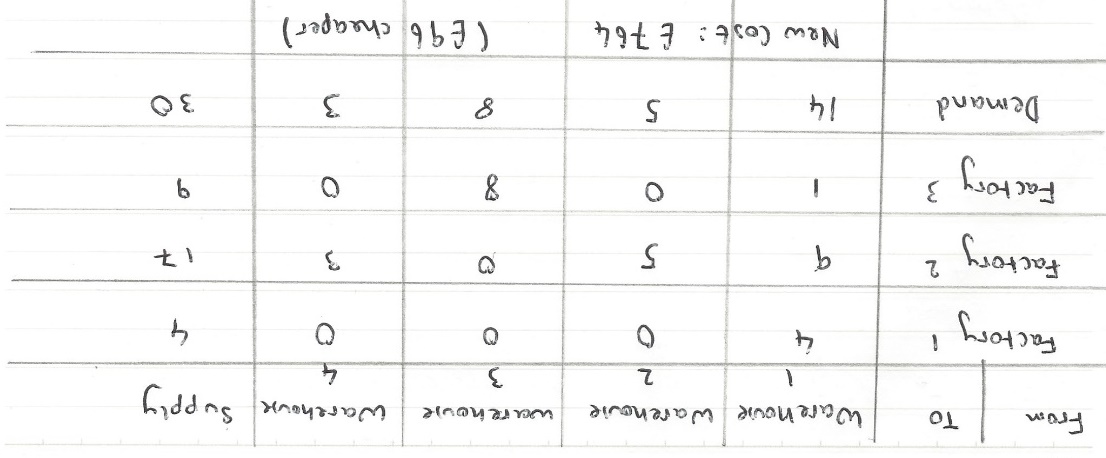
#the values of variables will be different so different sections of the code will be executed

END IF

END IF

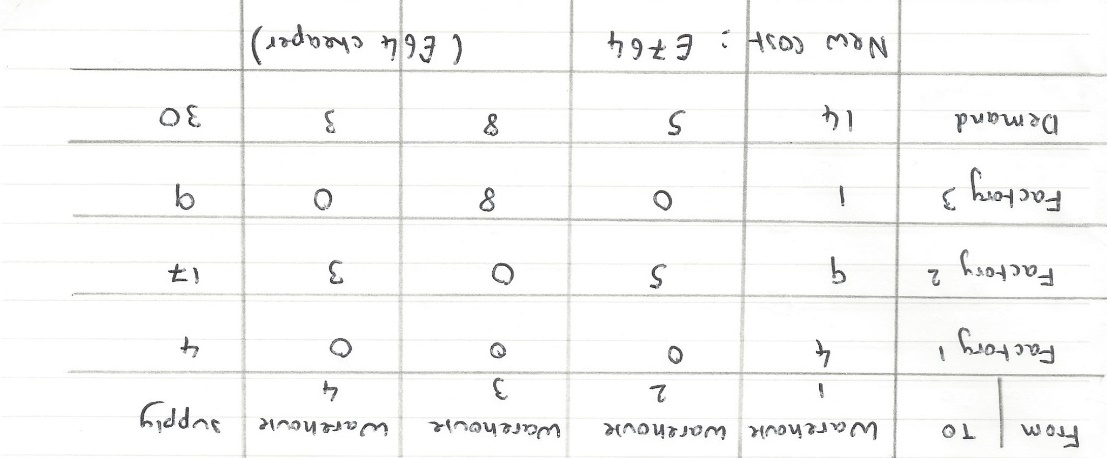
END IF

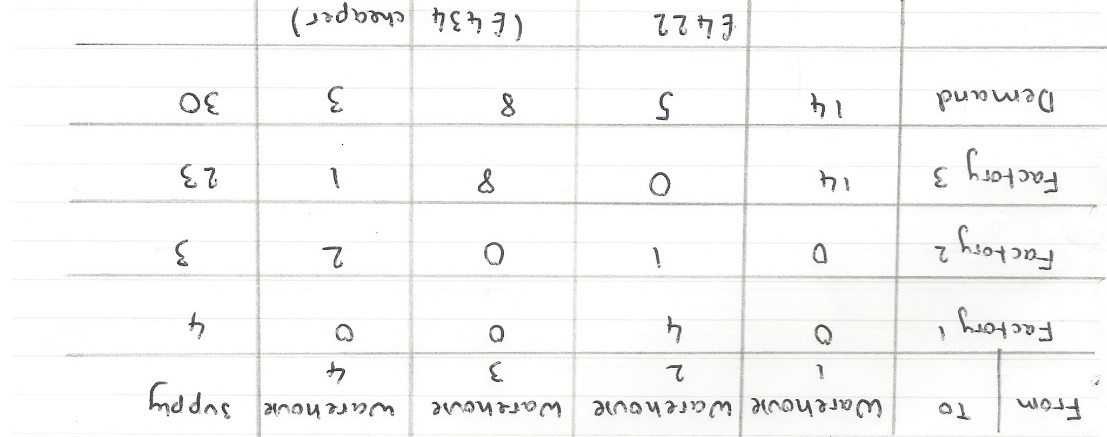
I carried out dry runs to optimise each method. The balanced results are shown below. I did not optimise the unbalanced matrices because they were all degenerate.

The North West Corner method:

The Least Cost Method:

Vogel’s Approximation Method:





## Techniques Used:

Throughout the code, all variables and function names have been given meaningful identifiers, so anyone reading the code can easily understand what the function/ variable is being used for. There are also comments indicating the use of a function and what different variables are being used for. This enhances the readability of the code and any comments that seem redundant demonstrate that the code is meeting its objective.

The code is contained within functions, being called by other functions. This avoids the repetition of code and divides the overall complex problem into simpler smaller problems. The use of functions also reduces the chances of error and makes the code easier to maintain. It also reduces the chance of any side effects in the code because it is all self-contained.

## Libraries:

I have used several libraries within my code to help implement the mathematical methods and to produce the user interface. This reduces the length of code that I need to write and reduces the time I need to spend on certain areas of the program. This means I can prioritise more complex areas, such as the optimisation algorithm.

The libraries I have used are:

* Tkinter – to use the widgets for my interface and to create the interface
* Math – to use the ‘inf’ data type as a comparison value when finding the lowest cost
* Copy – to create copies of the data arrays, so they can be manipulated whilst storing an original copy
* tableClass – a series of classes I wrote myself to aid the interface display

## Log of the stages of the investigation:

|  |  |  |
| --- | --- | --- |
| Stage | Time Spent | Explanation |
| Defining the investigation area | 5% | Establishing the problem that I will investigate. |
| Background research | 10% | Researching what the transportation model is and why there needs to be a solution to it. Also, researching the external factors affected by efficient transportation. |
| Requirements | 10% | Coming up with a list of all things the program needs to do in order to solve the transportation problem. As well as, the way the results need to be shown to the user in a way where the results of different algorithms can be produced. |
| Implementation | 30% | Programming the three main methods, the optimisation algorithm and classes needed to build the user interface. |
| Develop solution | 20% | Create any other functions that are needed to produce the intended program. Ensure the program code is efficient so there is no redundant code or data. |
| Test solution | 15% | Carry out numerous tests to ensure every part of the program works and it produces the correct result. This includes dry runs and runs on the computer. |
| Results | 10% | Use different data to produce results and compare the data produced for each method. Draw a conclusion from the results. |

# Technical Solution

The following pages contain the code I wrote to implement my investigation.

## Classes Created:

#table class for transportation model

import tkinter as tk

from tkinter import ttk, Canvas #importing the Tkinter libraries to be used to create the interface

#the Canvas library allows a programmer to create shapes and vector images

class tableClass:

def \_\_init\_\_(self, rows, columns, factories, warehouses, totalSupply, totalDemand, matrix, entries):

self.rows = rows

self.columns = columns

self.factories = factories

self.warehouses = warehouses

self.totalSupply = totalSupply

self.totalDemand = totalDemand

self.matrix = matrix

self.entries = entries

#initialising all variables needed throughout the class

def createTable(self):

#this method creates the entry matrix where the user can enter their data

if self.totalDemand != self.totalSupply:

total = self.totalDemand, ',', self.totalSupply

else:

total = self.totalSupply #creating a variable of the total supply/ demand that will be displayed on the interface

for y in range(self.rows):

row = []

for x in range(self.columns):

#loops through a number of times to get the right dimensions for the entry matrix

#there should be a column for each warehouse and a row for each factory

if y == 0 and x == 0:

a = ttk.Label(self.matrix)

#by creating a label, the user will not be able to change the text displayed

a["text"] = "Factory|Warehouse"

#creating a title to show what each row and column represents

row.append(None)

elif y == self.rows-1 and x == self.columns-1:

a = ttk.Label(self.matrix)

a["text"] = "Total:", total

#creating a label for the total supply and demand stated by the user

row.append(None)

elif x == 0 and y == (self.rows-1):

a = ttk.Label(self.matrix)

a["text"] = "Demand" #creating a title for the Demand column

row.append(None)

elif y == 0 and x == self.columns-1:

a = ttk.Label(self.matrix)

a["text"] = "Supply" #creating a title for the Supply row

row.append(None)

#all of the above cells are given a value of None so they aren't included in the 2D array used to store the user's data

else:

a = ttk.Entry(self.matrix)

#this enables the user to change/ input a value into the cell

if y == 0 and x > 0:

a.insert(0, "W" + str(x))

#prepopulates cell with label ('W1', 'W2' etc) but created so the user can change it

elif x == 0 and y > 0:

a.insert(0, "F" + str(y))

#prepopulates cell with label ('F1', 'F2' etc) but created so the user can change it

else:

pass

row.append(a) #adds the row just created to the 2D matrix

a["width"] = 14

# The length of the costs, supply and demand are not restricted as the length of them does not have an impact on the time taken for the program to run.

a.grid(row = y, column = x) #adding the cell to the Tkinter interface

self.entries.append(row) #adding the row created on the corresponding iteration to a single identifier

#this will form a 2D matrix

class analysisClass:

def \_\_init\_\_(self, cost, rows, columns, factories, warehouses, totalSupply, totalDemand, method, allocationArray, blankCanvas, degenerate):

self.cost = cost

self.rows = rows

self.columns = columns

self.factories = factories

self.warehouses = warehouses

self.totalSupply = totalSupply

self.totalDemand = totalDemand

self.method = method

self.allocationArray = allocationArray

self.blankCanvas = blankCanvas

self.degenerate = degenerate

#initialises variabales needed within this class



def \_\_calculateTotal(self):

#calculates the total allocations made for each row and column

#made as a private method so it cannot be called outside of the class

#this will ensure it is not confused with other functions in the main program

for x in range(self.factories):

totalAllocations = 0

for y in range(self.warehouses):

if self.allocationArray[x][y] > 0:

totalAllocations += self.allocationArray[x][y]

else:

pass

#iterates through the allocation array for each row and adds together the allocations made

self.allocationArray[x].append(totalAllocations)

#the allocations made are added to the end of each row of the alllcationArray

row = [0] \* self.columns

self.allocationArray.append(row)

#adds an row of zeros to the end of allocationArray so it can be edited later on

total = 0

for y in range(self.warehouses):

totalAllocations = 0

for x in range(self.factories):

if self.allocationArray[x][y] > 0:

#going through each column and adding together the allocations for that column

totalAllocations += self.allocationArray[x][y]

total += self.allocationArray[x][y]

else:

pass

self.allocationArray[-1][y] = totalAllocations

#apending the allocations made in the column to the end of the corresponding column

return total, self.allocationArray #returning the array and total so the appended versions can be used

def \_\_balancedChecked(self):

#to ensure dummy rows/ columns aren’t counted when creating the network graphs

if self.totalDemand > self.totalSupply:

del self.allocationArray[-1] #removing dummy row

elif self.totalSupply > self.totalDemand:

for j in self.allocationArray: #removing dummy column

del j[-1]

else:

pass

return self.allocationArray #returning updated allocationArray

def \_\_labelAllocationArray(self):

#adds the row and column labels to the allocation array

#another private method not to be called out of the class

label = ["F|W"]

#the labels for each row/ column will be added to the array before being added to the allocation array

for q in range(1, self.columns):

if q == self.columns-1:

label.append("TOTAL")

else:

label.append("W" + str(q))

#adding warehouse labels for each column, with the 'total' label in the last column

self.allocationArray.insert(0, label)

#inserting the label array into the allocationArray

for p in range(1, self.rows):

if p == self.rows-1:

label = "TOTAL"

else:

label = 'F'+str(p)

#adding factory labels to each row, with 'total' being added as the last row

self.allocationArray[p].insert(0, label)

#inserting labels for each row to the allocationArray so the user can identify what each cell means

return self.allocationArray #returning the updated allocation array

def analysisPage(self):

#adds the written information onto the results interface

self.blankCanvas.create\_text(150,10, text = "Using the "+ self.method +":")

self.blankCanvas.create\_text(150,30, text = "A network graph of the different routes taken.", fill = 'grey')

self.blankCanvas.create\_text(150,50, text = "The supplies to be transported are shown.", fill = 'grey')

self.blankCanvas.create\_text(150,80, text = "The total cost for this method is: £" + str(self.cost))

self.blankCanvas.create\_text(500,30, text = "This shows an adjacency matrix.\nThese are the units to be transported from the \nspecified factories to the specified warehouses", fill = "grey")

#outputting this information to the user so they can understand the results

if self.degenerate: #if degenerate the solution will not be optimised so the user must be told

self.blankCanvas.create\_text(150,65, text = "This is a degenerate solution.", fill = 'red')

def networkGraph(self):

#creates the network graph to be displayed on the results interface

nodes = [100, 200, 300, 400, 500, 600]

#each index will be used as a constant for the positioning of the graph nodes

#the maximum number of nodes there can be are 6

self.allocationArray = self.\_\_balancedChecked()

#checks whether or not the supply and demand are equal and if not adapts the allocation array so it is in the correct syntax

for x in range(self.factories):

self.blankCanvas.create\_oval(70, nodes[x], 100, nodes[x]+30, fill = "#aad6ff")

#the coordinates for each node changes because of the for loop, creating a line of nodes

self.blankCanvas.create\_text((85, nodes[x]+15), text="F"+str(x+1))

#labels each node with a factory number, the indexing starts at 0 so 1 must be added to the factory number

for y in range(self.warehouses):

self.blankCanvas.create\_oval(200, nodes[y], 230, nodes[y]+30, fill = "#ffb3ab")

self.blankCanvas.create\_text((215, nodes[y]+15), text="W"+str(y+1))

#creates nodes for all the warehouses and labels each one with the warehouse number

lineCreated = False #used as a fallback method if the solution has no allocations

for f in range(len(self.allocationArray)):

for w in range(self.warehouses):

if self.allocationArray[f][w] > 0:

#checking each cell to see if an allocation has been made

self.blankCanvas.create\_line(100,nodes[f]+15,200,nodes[w]+15)

#creates a line for an allocation between the corresponding factory (f) and warehouse (w)

lineCreated = True

else:

pass

if lineCreated == False:

#explains why there are no connections between nodes

self.blankCanvas.create\_text(500,60, text = "The solution is degenerate and unbalanced so there is no solution", fill = "red")

def resultsTable(self):

#creates the table showing the allocations

xCoOrd = [300,350,400,450,500,550,600,650,700]

yCoOrd = [100,130,160,190,220,250,280,310,340]

#both arrays are used to create coordinates for the table of allocations

total, self.allocationArray = self.\_\_calculateTotal()

#calculates the total allocations for each row and column and appends it to the allocationArray

self.allocationArray = self.\_\_labelAllocationArray()

#adds the row and column labels back in from the original input matrix

del self.allocationArray[-1][-1]

#removes very last cell as it is replaced in the previous line

for x in range(self.rows):

for y in range(self.columns):

self.blankCanvas.create\_rectangle(xCoOrd[y], yCoOrd[x], xCoOrd[y+1], yCoOrd[x+1])

#creates a table made up of rectangles

#it is the easiest way to create the table since the Canvas library is in use on this window

for x in range(self.rows):

for y in range(self.columns):

if x == self.rows-1 and y == self.columns-1:

self.blankCanvas.create\_text((xCoOrd[y]+25, yCoOrd[x]+15), text= str(total))

#adds total supply and demand label to the table

else:

self.blankCanvas.create\_text((xCoOrd[y]+25, yCoOrd[x]+15), text=str(self.allocationArray[x][y]))

#adds the allocations made at all factories/warehouses, if no allocations are made it is shown as 0

## Libraries used:

#transportation problem

import tkinter as tk

from tkinter import ttk, messagebox, Canvas

#used to create the user interface

import math

from math import inf

#used as a comparison value

import copy

from copy import deepcopy

#used to copy the data matrices so they can be manipulated whilst keeping an original copy

import tableClass

#written on a seprate program to make it easier to maintain

from tableClass import tableClass, analysisClass

#used to create the interface for both inputs and results

## Main Program:

root = tk.Tk() #creating an instance of Tk so my code can be interpreted

root.title("Transportation Model")

#it is a global variable because it is used in functions that are called by other functions

#this avoids passing paramters into functions that do not use them

#functions are used to separate the code for the main program

def validateMainWindow(factories, warehouses, totalDemand, totalSupply, window):

#validates all data inputs

#only shows the user one error at a time if there are multiple, so they are not overloaded with information

if factories < 0 or warehouses < 0 or totalDemand < 0 or totalSupply < 0:

messagebox.showerror(title = "ERROR", message = "You cannot have negative values.")

window.destroy() #stops the input window from opening

elif factories == 0 or warehouses == 0: #checks inputs are valid for the methods

messagebox.showerror(title = "ERROR", message = "There must be at least 2 factories and 2 warehouses.")

window.destroy()

elif factories == 1 or warehouses == 1:

messagebox.showerror(title = "ERROR", message = "There must be at least 2 factories and 2 warehouses. \nIf there is only 1 factory or 1 warehouse, then all items are sent from/ delivered to that factory/ warehouse.")

window.destroy()

elif totalDemand < warehouses or totalSupply < factories: #algorithms will not work unless these conditions are met

messagebox.showerror(title = "ERROR", message = "You must have at least " + str(factories) + " supply and " + str(warehouses) + " demand.")

window.destroy()

elif factories > 6 or warehouses > 6: #must limit size of table in order to reduce time taken for program to run

messagebox.showerror(title = "ERROR", message = "The maximum number of factories is 6, maximum number of warehouses is 6.")

window.destroy()

else:

pass

def matrixTable(factories, warehouses, totalSupply, totalDemand):

#collects data needed to create the entry matrix and then creates the matrix

entries = [] #this is where the user's data will be stored

window = tk.Toplevel(root) #creating a new window

window.title("Transportation Model")

matrix = tk.Frame(window) #creates a place hold where the input matrix will be

matrix.grid(row = 0, column = 0)

#places the widget onto the interface in a specific place

factories = factories.get()

warehouses = warehouses.get()

totalSupply = totalSupply.get()

totalDemand = totalDemand.get()

#collects the information inputted by the user on the interface

try:

factories = int(factories)

warehouses = int(warehouses)

totalSupply = int(totalSupply)

totalDemand = int(totalDemand)

#error checking that the user has inputted an item of data and it is a number

except:

messagebox.showerror(title = "ERROR", message = "You must enter integer numbers.")

#shows user suitable message if invalid input

window.destroy() #stops input window opening

validateMainWindow(factories, warehouses, totalDemand, totalSupply, window)

#runs other error checking conditions

rows = factories + 2

columns = warehouses + 2 #used to create layout of input matrix

tc = tableClass(rows, columns, factories, warehouses, totalSupply, totalDemand, matrix, entries)

#creates an instance of a class

tc.createTable() #calls a method of the tableClass, which creates the input matrix

menuOptions(window, matrix, rows, columns, entries, totalSupply, totalDemand, factories, warehouses)

#calls next function to be executed

def menuOptions(window, matrix, rows, columns, entries, totalSupply, totalDemand, factories, warehouses):

#creates the menu options to be displayed

options = tk.Frame(window)

options.grid(row = 0, column = 1) #creates new placehold where the menu will be shown

methodMenu = ttk.Menubutton(options, text = "Methods") #creates initial button for the methods

methodMenu.menu = tk.Menu(methodMenu) #initialises the menu widget

methodMenu.menu["title"] = "Methods"

methodMenu["menu"]= methodMenu.menu #sets the property of menu button so it is associated with the method menu

##found on:

##https://www.tutorialspoint.com/python3/tk\_menubutton.htm

methodMenu.menu.add\_command(label = "Least Cost", command = lambda: leastCost(entries, rows, columns, totalSupply, totalDemand , factories, warehouses, window))

methodMenu.menu.add\_command(label = "Vogel's Approx.", command = lambda: vogelsApprox(entries, rows, columns, totalSupply, totalDemand, factories, warehouses, window))

methodMenu.menu.add\_command(label = "North West Corner", command = lambda: northWestCorner(entries, rows, columns, totalSupply, totalDemand, factories, warehouses, window, matrix))

methodMenu.grid(row = 0, column = 1, sticky = "N", padx = 5, pady = 5)

#calling selected 'method' if user clicks on one of the menu options

helpChoices = tk.Frame(window)

helpChoices.grid(row = 0, column = 2)

helpMenu = ttk.Menubutton(helpChoices, text = "Help")

helpMenu.menu = tk.Menu(helpMenu)

helpMenu.menu["title"] = "Help"

helpMenu["menu"] = helpMenu.menu

helpMenu.menu.add\_command(label = "Matrix Help", command = lambda: helpMatrix())

helpMenu.menu.add\_command(label = "Methods Help", command = lambda: helpMethods())

helpMenu.grid(row = 0, column = 2)

#creates another drop down menu widget with extra information to understand the interface

def validateInputs(rawData, rows, columns, factories, warehouses, window, totalSupply, totalDemand):

#validates data entered into the entry matrix

totalZero = 0 #used to check if allocationarray is empty

totalValue = 0 #used to check if data is all the same

matrixSupply = 0

#used to check if the total supply entered is equal to the one stated in the first window

matrixDemand = 0

#used to check if the total demand inputted is the same as in the first window

for i in range (factories):

for j in range(warehouses):

if rawData[i][j] == 0:

totalZero += 1

#incrementing the count by 1 if an input is zero

#used for validation of inputs later

elif rawData[i][j] == rawData[0][0]:

totalValue += 1

#checking if an input is the same as the first piece of data entered

#used for validation

else:

pass

if totalSupply == totalDemand:

Factories = factories + 1

Warehouses = warehouses + 1

elif totalSupply < totalDemand:

Factories = factories #ensures dummy data is ignored in final solution

Warehouses = warehouses + 1

else:

Factories = factories + 1

Warehouses = warehouses

#finding the number of rows/columns to be included when adding total demand/supply

for x in range(Factories):

matrixSupply += rawData[x][-1]

#adding up all supplies entered in the input table

for y in range(Warehouses):

matrixDemand += rawData[-1][y]

#adding up all demands entered in the input table

if matrixSupply != totalSupply:

#ensuring data in the entry matrix matches the data entered in the first window

messagebox.showerror(title = "ERROR", message = "Your supplies do not add up to the total you entered.")

return False

elif matrixDemand != totalDemand:

messagebox.showerror(title = "ERROR", message = "Your demands do not add up to the total you entered.")

return False

else:

pass

if totalZero == (factories\*warehouses):

#ensuring the user hasnt entered only zeros

messagebox.showerror(title = "ERROR", message = "You must enter several costs above zero or all routes will be free, so there is no optimum solution.")

return False

elif totalValue == (factories\*warehouses):

#do not need to run algorithm if the cost is the same everywhere

#it is very unlikely for this to happen

messagebox.showerror(title = "ERROR", message = "When all values are the same, there is no optimal solution, all routes will have the same outcome.")

return False

else:

return True

#due to the way Tkinter works, i cannot validate this data until the user has chosen a method

#so this validation cannot be executed at the same time as any previous error checks

def collectData(entries, rows, columns, totalSupply, totalDemand, factories, warehouses, window):

#collects the data entered in the matrix on the interface

data = [] #used to collect data inputted by user

allocationArray = [] #will be used to create layout of matrix

for i in range(rows):

row = [None] \* columns

#creates array of empty cells- to create initial structure of the data array

data.append(row)

if i > 0 and i < rows-1:

allocationArray.append([0] \* (warehouses))

#creates array of zeros for initial structure of the allocation array

for j in range(columns):

if entries[i][j] is not None and entries[i][j].get():

#if a cell's value is set to None it's value is added later

#the user wasnt allowed to enter data into these cells

if i > 0 and j > 0:

try:

data[i][j] = int(entries[i][j].get())

#tries to collect integer data input by the user from the interface

if data[i][j] < 0:

#you cannot have negative costs, so if the user enters them an appropriate error message will be shown

messagebox.showerror(title = "ERROR", message = "You must only enter numbers above zero.")

return data, rows, columns, allocationArray, False

#False will stop the next part of the program running

else:

pass

except:

messagebox.showerror(title = "ERROR", message = "You must only enter whole numbers for the costs, supply and demand.")

#validates the user input and displays appropriate error if incorrect

return data, rows, columns, allocationArray, False

else:

data[i][j] = str(entries[i][j].get())

#if it is in the first row or column the value is stored as a string as it represent the names of the factories and warehouses

else:

if entries[i][j] is not None and len(entries[i][j].get()) == 0:

messagebox.showerror(title = "ERROR", message = "You must enter a value into every empty cell.")

return data, rows, columns, allocationArray, False

else:

data[i][j] = 0

#changes values not inputted to zero ie, the values in the table that were not avliable for the user to edit

if totalDemand < totalSupply:

#creating a dummy column so the algorithms can be implemented as the current data is unbalanced

columns += 1

balance = totalSupply - totalDemand #to be placed in the dummy column

for q in range(rows):

data[q].append(data[q][-1])

data[q][-2] = 0

#moving the demands column across a column, and inserting a column of zeros where the demands column previously was

#the column of zeros is to be used as a dummy column

for p in range(factories):

allocationArray[p].append(allocationArray[p][-1])

#adding another column of zeros to the array of zeros

data[-1][-2] = balance

#adding the value needed to balance supply and demand to the dummy column

elif totalSupply < totalDemand:

#creating a dummy row for unbalanced matrices

rows += 1

balance = totalDemand - totalSupply

demands = deepcopy(data[-1])

data.append(demands)

data[-2] = [0] \* columns

#creating two duplicate rows of the demands and replacing the first row with zeros

#this will be the dummy row

data[-2][-1] = balance

#changing the final demand column to the value needed to balance supply and demand

lastRow = deepcopy(allocationArray[-1])

allocationArray.append(lastRow)

#adding the row of zeros on to the allocation array to act as a dummy row

else:

pass

#if the supply and demand are equal, the matrix is balanced and no changes need to be made

rawData = deepcopy(data)

#the copy will be used to create a matrix without the row and column labels

del rawData[0] #removes first row of warehouse labels

for j in rawData: #removes factory labels

del j[0]

run = validateInputs(rawData, rows, columns, factories, warehouses, window, totalSupply, totalDemand)

#check all inputs are valid so there will not be any errors later on

return rawData, rows, columns, allocationArray, run

#returns all updated variables to be used in the method algorithms

def identicalMatrices(factories, warehouses, allocationArray, finalAllocations):

#checking if the original solution is identical to the optimal solution

#this is possible due to the way the stepping stone algorithm works

del finalAllocations[0]

for j in finalAllocations:

del j[0]

#the labels were added to the finalAllocations array, so must be removed before comparison

count = 0

#used to keep track of how many cells are identical

for x in range(factories):

for y in range(warehouses):

#loops through both matrices to compare values

if allocationArray[x][y] == finalAllocations[x][y]:

count += 1

#if the cells are identical the count is incremented by 1

else:

pass

if count == factories\*warehouses:

#if all cells are identical

return True

else:

return False #they are not identical

def costCalculate(factories, warehouses, costArray, allocationArray):

#calculates the overall cost of a solution

cost = 0

for y in range(factories):

for x in range(warehouses):

if allocationArray[y][x] > 0:

#loops through the allocation array and finds where an allocation has been made

cost += allocationArray[y][x] \* costArray[y][x]

#multiplies the cost per unit by the number of allocations made

#adds the costs of all allocations together to find the overall cost

return cost

def resultsWindow(cost, method, rows, columns, factories, warehouses, totalSupply, totalDemand, entries, finalAllocations, degenerate):

#creates a results page for the data passed to the function

windowPage = tk.Tk()

#creates an instance of Tk to create another window

#this will be separate to the other windows so cannot be closed using the same command

windowPage.title(method)

#shows the user which method the results are for

blankCanvas = Canvas(windowPage, width = 700, height = 650)

#uses the Tkinter Canvas widgit so the graphs can be drawn

blankCanvas.grid(row = 0, column = 0)

analysis = analysisClass(cost, rows, columns, factories, warehouses, totalSupply, totalDemand, method, finalAllocations, blankCanvas, degenerate)

#creates an instance of the analysisClass so the methods can be called

analysis.analysisPage()

#creates an output for all written information to go onto the results page

analysis.networkGraph()

#creates the network graph for the particular instance

analysis.resultsTable()

#creates a table of all the allocations made

def numLocationsUsed(factories, warehouses, allocationArray):

#calculates the number of locations where allocations have been made

locations = 0

#reset every time the function is called

for x in range(factories):

for y in range(warehouses):

if allocationArray[x][y] > 0:

#iterates through the allocation array and counts the number of places where allocations were made

locations += 1

return locations

#returns the number counted

def originalSolution(root, window, factories, warehouses, costArray, allocationArray, rows, columns, totalSupply, totalDemand, entries, degenerate, method):

#creates a window showing the initial feasible solution

solutionWindow = tk.Toplevel(root)

#creates the window

solutionWindow.title("Original Solution")

initialSolutionTable = tk.Frame(window)

initialSolutionTable.grid(row = 0, column = 0)

#creates a placehold for the initial solution table

blankCanvas = Canvas(solutionWindow, width = 700, height = 650)

#uses the Tkinter library Canvas, so the graphs can be drawn

blankCanvas.grid(row = 0, column = 0)

if isinstance(allocationArray[1][0], str):

#checks if the allocation array has labels for the rows and columns

del allocationArray[0]

#removes first row of warehouse labels

for j in allocationArray: #removes factory labels

del j[0]

cost = costCalculate(factories, warehouses, costArray, allocationArray)

#calculates the cost of the initial solution

initialSolution = analysisClass(cost, rows, columns, factories, warehouses, totalSupply, totalDemand, method, allocationArray, blankCanvas, False)

#creates an instance of the class so the methods can be called

initialSolution.analysisPage()

initialSolution.networkGraph()

initialSolution.resultsTable()

#produces the results page with all the information on it for the original solution

def northWestCorner(entries, rows, columns, totalSupply, totalDemand, factories, warehouses, window, matrix):

rawData, rows, columns, allocationArray, run = collectData(entries, rows, columns, totalSupply, totalDemand, factories, warehouses, window)

#obtains data from the input page to be used in the algorithm

if run == False:

return

#to stop function running any more of the algorithm- if the user's data is invalid

northWCAllocations = deepcopy(allocationArray)

#taking a copy of the allocation array so it can be manipulated whilst keeping an original copy

costArray = deepcopy(rawData)

#rows is x, columns is y

y = 0

x = 0

#sets inital cell reference

supply = rawData[x][-1]

#calculates supply for current factory

allocations = 0

#used to count the total allocations made as the algorithm runs, used as a condition for the execution of the algorithm

demand = rawData[-1][y]

#calculates demand for current warehouse

while not(allocations == totalSupply) and not(allocations == totalDemand):

#code is run until all possible allocations have been made

if demand == supply:

northWCAllocations[x][y] = demand

#demand and supply are balanced so the maximum allocations can be made to the current cell

allocations += supply

y += 1

x += 1

#no more allocations possible for the current row or column, so x and y are both incremented

supply = rawData[x][-1]

demand = rawData[-1][y] #the supply and demand of the next cell are found and stored

elif demand < supply:

northWCAllocations[x][y] = demand

#demand is smaller than supply, so the maximum allocations to be made will be the demand value

allocations += demand

supply -= demand #the remaining supply is calculated

y += 1

#there is supply remaining so another allocation possible so x stays the same

demand = rawData[-1][y] #the demand of the next cell is found

else:

northWCAllocations[x][y] = supply

#the supply is larger than demand, so the maximum allocations possible will be the supply value

demand -= supply

#the remaining demand is calculated

allocations += supply

#the allocations made are added to the running total

x += 1

#there is no more supply on the current row, so the next cell will be on the row below

supply = rawData[x][-1] #the new supply is stored

allocationArray = deepcopy(northWCAllocations)

if totalDemand == totalSupply:

cost = costCalculate(factories, warehouses, costArray, allocationArray)

#calculating cost of initial feasible solution

elif totalDemand > totalSupply:

rows -= 1

#not taking into account the dummy cells as they were added for the syntax of the algorithm

cost = costCalculate(factories, warehouses, costArray, allocationArray)

else:

columns -= 1

cost = costCalculate(factories, warehouses, costArray, allocationArray)

allocations = numLocationsUsed(factories, warehouses, allocationArray)

#returns the number of places where allocations are made in the initial feasible solution

finalAllocations, cost, degenerate = steppingStone(rows, columns, window, factories, warehouses, allocationArray, rawData, costArray, allocations, totalSupply, totalDemand)

#optimisation algorithm executed on the initial feasible solution

method = "North West Corner"

#used as a parameter in the results function

if finalAllocations == None:

finalAllocations = allocationArray

#original solution used if optimisation algorithm cannot be executed

if cost == None:

cost = costCalculate(factories, warehouses, costArray, allocationArray)

#cost of original solution needs to be recalculated if the optimisation algorithm was not executed fully ie, because the solution is degenerate

original = messagebox.askyesno(title = "Results", message = "Would you like to see the original solution as well as the optimised solution?")

resultsWindow(cost, method, rows, columns, factories, warehouses, totalSupply, totalDemand, entries, finalAllocations, degenerate)

#results page is called once final solution has been calculated to show the user the results

if original:

#the initial feasible solution is shown on a new window if the user wants to see it

identical = identicalMatrices(factories, warehouses, allocationArray, finalAllocations)

#returns whether or not the original and optimal solution are the same

if degenerate:

messagebox.showinfo(title = "Original solution", message = "The solution is degenerate so the original solution is the same as the optimum solution.")

elif identical:

#if the matrices are the same, it only needs to be shown once

messagebox.showinfo(title = "Original Solution", message = "The original solution and optimal solution are the same.")

else:

originalSolution(root, window, factories, warehouses, costArray, allocationArray, rows, columns, totalSupply, totalDemand, entries, degenerate, method)

else:

pass

def leastCost(entries, rows, columns, totalSupply, totalDemand, factories, warehouses, window):

rawData, rows, columns, allocationArray, run = collectData(entries, rows, columns, totalSupply, totalDemand, factories, warehouses, window)

#collecting data from user input

if run == False:

return

#algorithm will not run if the input matrix is invalid

allocations = 0

#used as a condition as to whether or not the algorithm is executed

data=rawData

costArray = deepcopy(data)

#the data array is edited in this algorithm so a copy must be taken in order to be used later on

columns -= 1

rows -= 1

#algorithm doesn't take into account the supply and demand row/ column when finding the lowest cost

if totalDemand < totalSupply:

#finds the total allocations that will be made by using the highest value from the supply and demand

totalAllocationsNeeded = totalSupply

else:

totalAllocationsNeeded = totalDemand

while allocations < totalAllocationsNeeded:

#algorithm will run until all allocations possible have been made

value = inf

#used as an initial comparison to find the lowest value in the input matrix

for x in range(rows-1):

for y in range(columns-1):

if data[x][y] < value:

#iterates through matrix and compares every cost to the value variable

xIndex = x

#if a cost lower than 'value' is found, the coordinates of the lower value are stored

yIndex = y

value = data[x][y]

#the lower value becomes the new comparison value

elif data[x][y] == value:

if data[x][y] != inf:

#if the two comparison values are both infinite, no changes need to be made as one value is not lower than the other

currentSupply = data[xIndex][-1]

currentDemand = data[-1][yIndex]

#finds the supply and demand of the lowest value

newSupply = data[x][-1]

newDemand = data[-1][y]

#finds the supply and demand of the same value but in a different cell

if newSupply > currentSupply:

#if more items can be moved then the new cell location should be stored instead of the current one

xIndex = x

yIndex = y

elif newDemand > currentDemand:

#if the above conditions aren't met, but the new cell location has a larger demand, then the new cell location should be stored instead of the current one

xIndex = x

yIndex = y

else:

pass

else:

pass

else:

pass

#if these conditions are not met, no changes will need to be made

supply = data[xIndex][columns-1]

demand = data[-1][yIndex]

#finds the supply and demand of the cell with the lowest cost

if supply == demand:

allocationArray[xIndex][yIndex] = demand

#using the supply/ demand constraint to calculate the maximum allocations that can be made

allocationArray[xIndex][yIndex] = demand

#using the supply/ demand constraint to calculate the maximum allocations that can be made

allocations += supply

#adding the allocations made to the total allocation count

for x in range(rows-1):

data[x][yIndex] = inf

for y in range(columns-1):

data[xIndex][y] = inf

#changing the row and columns of the cell where the allocation was made to infinite as no more allocations can be made

#by changing them to infinite, the costs will no longer be compared when finding the lowest cost

data[xIndex][-1] = 0

data[-1][yIndex] = 0

#changing the supply and demand of the row and column to zero as no more items need to be dispensed/ allocated

elif demand < supply:

allocationArray[xIndex][yIndex] = demand

#the demand is lower so that is the maximum number of allocations that can be made

allocations += demand

supply -= demand

#there will still be supply remaining, so this must be calculated and updated in the data array

for x in range(rows-1):

data[x][yIndex] = inf

#all demands have been filled so the column of the allocation cell cannot be used as possible allocation locations

data[-1][yIndex] = 0

#changing the demand to show it has all been used

data[xIndex][-1] = supply

#updating the supply requirements so allocations can be made

else:

allocationArray[xIndex][yIndex] = supply

#the supply is lower than demand if the code is in this if statement

demand -= supply

#the supply will all be used up, leaving some demand at the current warehouse

allocations += supply

data[xIndex][-1] = 0

data[-1][yIndex] = demand

#updating the demand in the data array to ensure allocations are made

for y in range(columns-1):

data[xIndex][y] = inf

#all supply at this factory has been used so the row can no longer be used to make allocations

rows += 1

columns += 1

#so the costCalculate function runs through the entire matrix

if totalDemand == totalSupply:

cost = costCalculate(factories, warehouses, costArray, allocationArray)

elif totalDemand > totalSupply:

rows -= 1

#not taking into account the dummy column when calculating cost

cost = costCalculate(factories, warehouses, costArray, allocationArray)

else:

columns -= 1

#not taking into account the dummy row

cost = costCalculate(factories, warehouses, costArray, allocationArray)

rawData = deepcopy(costArray)

#the rawData array was changed so a copy of the original data array must be taken in order to be used later on

allocations = numLocationsUsed(factories, warehouses, allocationArray)

#returns the number of locations where allocations were made

finalAllocations, cost, degenerate = steppingStone(rows, columns, window, factories, warehouses, allocationArray, rawData, costArray, allocations, totalSupply, totalDemand)

#optimises the initial feasible solution and returns the optimised result

method = "Least Cost"

#used to show the name of the algorithm when the results are shown to the user

if finalAllocations == None:

#if solution is degenerate, the initial solution will be used

finalAllocations = allocationArray

if cost == None:

#if the solution is degenerate the cost and finalAllocatiosn will be set to None, so the cost of the initial solution will need to be recalculated

cost = costCalculate(factories, warehouses, costArray, allocationArray)

original = messagebox.askyesno(title = "Results", message = "Would you like to see the original solution as well as the optimised solution?")

resultsWindow(cost, method, rows, columns, factories, warehouses, totalSupply, totalDemand, entries, finalAllocations, degenerate)

#results are shown to the user using tkinter

if original:

#the initial feasible solution is shown on a new window

identical = identicalMatrices(factories, warehouses, allocationArray, finalAllocations)

if degenerate:

messagebox.showinfo(title = "Original solution", message = "The solution is degenerate so the original solution is the same as the optimum solution.")

elif identical:

#the solution only needs to be shown once

messagebox.showinfo(title = "Original Solution", message = "The original solution and optimal solution are the same.")

else:

originalSolution(root, window, factories, warehouses, costArray, allocationArray, rows, columns, totalSupply, totalDemand, entries, degenerate, method)

else:

pass

def calculateDifference(orderedArray):

#calculates the difference of the two lowest costs in an array passed as a parameter

if len(orderedArray) == 1:

#if no allocations can be made on the column/ row then only one cost will be avaliable so that is the difference

difference = orderedArray[0]

elif len(orderedArray) > 1:

difference = orderedArray[1] - orderedArray[0]

else:

#called when no allocations can be made

difference = 0

#the row/ column with the largest difference is where an allocation is made, so this means any allocation that could be made, will be made

return difference

def findDifferences(factories, warehouses, data):

#the difference of the two lowest costs is found for each row and column

for x in range(factories):

orderedArray = []

#creates new array for each row

for y in range(warehouses):

if data[x][y] != inf:

#if there is a cost not equal to infinity then it is appended to the row

orderedArray.append(data[x][y])

else:

pass

orderedArray.sort()

#uses the built in function to order the costs from lowest to highest

data[x][-1] = calculateDifference(orderedArray)

#adds the difference of that row returned from the function to the last column

#this column was appended to the data array and originally set to all zeros

for y in range(warehouses):

orderedArray = []

#creates new array for each column

for x in range(factories):

if data[x][y] != inf:

#if the cost is not infinite, it is added to the array

orderedArray.append(data[x][y])

else:

pass

orderedArray.sort()

#array is sorted into ascending order

data[-1][y] = calculateDifference(orderedArray)

#the difference for the current column is added to the last row of the data array

#this row was appended before the function was run

return data #the updated data array is returned to the Vogel's function

def findRowLowestCost(x, warehouses, data):

#takes in the x coordinate of the current row

#finds the cell with the lowest cost for the specified row

orderedArray = []

for y in range(warehouses):

if data[x][y] != inf:

orderedArray.append(data[x][y])

#finds the costs of the current row and appends them to an empty array

else:

pass

if len(orderedArray) == 0:

#this means there are no costs that aren't infinite so no more allocations can be made

return inf

#this will ensure no allocations will try to be made for the current row

else:

orderedArray.sort()

return orderedArray[0]

#returns the lowest cost from the current row

#if this row has the highest difference, an allocation will be made at this cell

def highestRowDifference(data, factories, warehouses, columns):

#finding the row with the highest difference in the two lowest costs

currentDifference = 0

#trying to find the highest difference so this will be used for comparison

xIndex = None

#will be used to check if a previous xIndex has been stored

count = None

#used as a point of reference in a future function

#this function is when count = 0, the highestColumnDifference is when count = 1

for x in range(factories):

if data[x][-1] > currentDifference:

#goes through the data array and compares the differences of all the rows

currentDifference = data[x][-1]

#wants the highest difference, so if the current row difference is higher, it will become the new comparison value

xIndex = x

#the row with the higher difference is stored so it can be used in the main section of Vogel's

count = 0

#to let the program know the highest difference is a row

elif data[x][-1] == currentDifference:

if xIndex == None:

#if there hasn't been a previous xIndex stored, it will store the current one

xIndex = x

else:

lowCostRowOne = findRowLowestCost(x, warehouses, data)

#finds the lowest cost on this row

lowCostRowTwo = findRowLowestCost(xIndex, warehouses, data)

#finds the lowest cost of the previously stored row

if lowCostRowOne < lowCostRowTwo:

#if the cost is lower, the items should be allocated here instead of at the previous row

xIndex = x #storing the new row

else:

pass

else:

pass

if xIndex == None:

#this is when all rows have a difference of zero so it will not make a difference which row is chosen

xIndex = 0

return currentDifference, xIndex, count

#this information will be needed by the main section of Vogel's if there is not a lower difference at one of the columns

def findColumnLowestCost(y, factories, data):

#takes the y coordinate of the current column being worked on

#finds the cell with the lowest cost

orderedArray = []

for x in range(factories):

if data[x][y] != inf:

#finds all costs in the column that are not infinite and appends them to an array

orderedArray.append(data[x][y])

else:

pass

if len(orderedArray) == 0:

#this is when all costs are infinite so no allocations will be made

return inf

else:

orderedArray.sort()

#the cell with the lowest cost is found and returned

return orderedArray[0]

def highestColumnDifference(data, xIndex, currentDifference, factories, warehouses, count):

#finds the column with the highest difference

yIndex = None

#similar to xIndex, this variable will be used to check if a previous value has been stored

for y in range(warehouses):

#iterates through the columns comparing the differences of each

if data[-1][y] > currentDifference:

#the largest difference will be used

#the column differences are compared to the highest row difference from the previous function until a higher difference is found

#if no difference is found to be higher than the row difference, then the highest row difference is used in the algorithm

currentDifference = data[-1][y]

#the higher difference becomes the comparison value

yIndex = y

count = 1

#lets Vogel's algorithm identify that the highest difference is a column

elif data[-1][y] == currentDifference:

if yIndex == None:

#no previous column index has been stored, so the highest difference is a row

lowCostRow = findRowLowestCost(xIndex, warehouses, data)

#finding the lowest cost for the row

lowCostColumn = findColumnLowestCost(y, factories, data)

#finding the lowest cost for the current column being stored

if lowCostColumn < lowCostRow:

#need to use the row/column with the lowest cost

yIndex = y

#storing the current column to be referenced to later on

count = 1

#updated to let the program know the highest difference is now a column

else:

pass

else:

#if the previous highest difference is a column

lowCostColumnOne = findColumnLowestCost(y, factories, data)

lowCostColumnTwo = findColumnLowestCost(yIndex, factories, data)

#finding the lowest cost for both columns

if lowCostColumnOne < lowCostColumnTwo:

#the lowest cost is where the allocation will be made

yIndex = y

#new column index is stored

else:

pass

else:

pass

return count, xIndex, yIndex

#values will be stored in both the xIndex and yIndex depending on where the highest difference is

#the value of 'count' will determine whether the xIndex or yIndex is used

def vogelsApprox(entries, rows, columns, totalSupply, totalDemand, factories, warehouses, window):

rawData, rows, columns, allocationArray, run = collectData(entries, rows, columns, totalSupply, totalDemand, factories, warehouses, window)

#the data inputted by the user is collected from the interface and stored in the values returned by the function

if run == False:

#stops the algorithm running if there is an error with the inputted data

return

data=rawData

#name is changed for consistency, as the identifier 'data' is used in other functions

costArray = deepcopy(data)

#the data array will be changed so a copy is made to be used later

data = [x + [0] for x in data]

#adding an extra column onto the matrix where the differences will be stored

data.append([0]\*columns)

#adding an extra row onto the matrix where the differences will be stored

allocations = 0

#used as a condition as to whether or not the algorithm executes another time

while not(allocations == totalSupply) and not(allocations == totalDemand):

data = findDifferences(factories, warehouses, data)

#returns the function with all differences calculated

currentDifference, xIndex, count = highestRowDifference(data, factories, warehouses, columns)

#finds the row with the highest difference

count, xIndex, yIndex = highestColumnDifference(data, xIndex, currentDifference, factories, warehouses, count)

#if a column has a higher difference than the one found in the rows function, the index of the column will be used

lowestCost = inf

#used as a comparison when finding the lowest cost of a row

if count == 0:

#if highest difference is a row then this code will be executed

for p in range(warehouses):

if data[xIndex][p] < lowestCost:

#finds the y coordinate of the cell with the lowest cost

lowestCost = data[xIndex][p]

#the xIndex is the value when the highest difference was found

yIndex = p

#iterates through the row to find the lowest cost

else:

pass

else:

#if the highest difference is a column then the following code will be executed

for r in range(factories):

#iterates through the column to find the lowest cost

if data[r][yIndex] < lowestCost:

#the yIndex was previosuly found when finding the highest difference

lowestCost = data[r][yIndex]

xIndex = r

#the xIndex of the cell with the lowest cost is stored

else:

pass

#the coordinates of the cell with the lowest cost, on the row/ column with the highest difference has now been stored

supply = data[xIndex][-2]

demand = data[-2][yIndex] #finding the supply and demand of the selected cell

if supply == demand:

allocationArray[xIndex][yIndex] = supply

#as the supply and demand are balanced, it can all be allocated to the current cell

allocations += supply

#the allocation just made is added to the running total

supply = 0

demand = 0

#all supply and demand will be used up

for r in range(rows):

data[r][yIndex] = inf

for s in range(columns):

data[xIndex][s] = inf

#the cost of the cells in the row and column of the current cell are set to infinite

#so they are not used when finding the next cell coordinates

data[xIndex][-2] = 0

data[-2][yIndex] = 0

#updating the supply and demand constraints in the data array

elif demand < supply:

allocationArray[xIndex][yIndex] = demand

#the demands can be fulfilled so are allocated to the selected cell

allocations += demand

supply -= demand

#the remaining supply is calculated

for r in range(rows):

data[r][yIndex] = inf

#the costs of the column are changed to infinite as there is no demand left

data[xIndex][-2] = supply

data[-2][yIndex] = 0

#supply and demand are updated so they can be used in the next iteration

else:

allocationArray[xIndex][yIndex] = supply

#the supply can be used up

demand -= supply

#calculating the remaining demand for the selected column

allocations += supply

for s in range(columns):

data[xIndex][s] = inf

#there is no supply left on the selected row so all costs must be changed to infinite

data[-2][yIndex] = demand

data[xIndex][-2] = 0

#the supply and demand constraints for the selected row/ column are updated in the data array

if totalDemand == totalSupply:

cost = costCalculate(factories, warehouses, costArray, allocationArray)

#the cost of the initial feasbile solution is calculated

elif totalDemand > totalSupply:

rows -= 1

#updating the number of rows so the dummy row isnt taken into account

cost = costCalculate(factories, warehouses, costArray, allocationArray)

else:

columns -= 1

#updating the variable so the dummy column isnt taken into account

cost = costCalculate(factories, warehouses, costArray, allocationArray)

#the current allocations variable has the total number of allocations

#the number of locations where allocations have been made is needed

allocations = numLocationsUsed(factories, warehouses, allocationArray)

#this returns the number of lcoations

finalAllocations, cost, degenerate = steppingStone(rows, columns, window, factories, warehouses, allocationArray, rawData, costArray, allocations, totalSupply, totalDemand)

#the initial solution is then passed through this function to optimise it, and the resulting data is returned

if finalAllocations == None:

#if solution is degenerate, the original solution is used

finalAllocations = allocationArray

if cost == None:

#if solution is degenerate then the cost will need to be recalculated

cost = costCalculate(factories, warehouses, costArray, allocationArray)

method = "Vogel's Approximation Method"

#used in following function to be printed on the results page

original = messagebox.askyesno(title = "Results", message = "Would you like to see the original solution as well as the optimised solution?")

resultsWindow(cost, method, rows, columns, factories, warehouses, totalSupply, totalDemand, entries, finalAllocations, degenerate)

#results are shown to the user using tkinter

if original:

#the initial feasible solution is shown on a new window

identical = identicalMatrices(factories, warehouses, allocationArray, finalAllocations)

if degenerate:

#if optimal solution is degenerate, the original solution is used as the final solution

messagebox.showinfo(title = "Original solution", message = "The solution is degenerate so the original solution is the same as the optimum solution.")

elif identical: #the original and optimal solution might be identical, so it only needs to be shown once

messagebox.showinfo(title = "Original Solution", message = "The original solution and optimal solution are the same.")

else:

originalSolution(root, window, factories, warehouses, costArray, allocationArray, rows, columns, totalSupply, totalDemand, entries, degenerate, method)

else:

pass

def degeneracyCheck(factories, warehouses, allocations):

#checks whether a solution is degenerate

if allocations < factories+warehouses-1:

#degeneracy formula found in my research

return True

else:

return False

#Boolean variable used to determine the outcome of the chosen algorithm

def pathFound(vertices):

#checks if a path has been found form the current vertices

if vertices[-1][0] == vertices[0][0]:

path = True

#a Boolean variable used to determine whether or not a function is run

elif vertices[-1][1] == vertices[0][1]:

#if it is on the same row/ column the path is complete

path = True

else:

path = False

#if not then the function is run again

return path

def rowSearch(x, y, allocationArray, vertices, columns, rows, skipVertex, factories, dummySource, recurse, count, changeY, timer, warehouses):

#contains parameters that depend upon the results of the column search

vertexFound = False

if recurse == True:

count += 1

#used in the column search to keep track of how many times the functions have been called without finding a vertex

if count > factories:

#if its been recursed more times than the number of factories, the previous two vertices should be removed

count = 0

#set back to zero as the vertices will be changed

vertexFound = True

timer += 1

if timer >= 2:

#the timer shouldnt go above 3 or the vertex is invalid, ie, a path cannot be found from the current vertex

changeY = True

#changes the Boolean conditions, so a different section of the code is run

timer = 0

#set back to zero as the vertices will be changed

else:

vertices.pop()

vertices.pop()

#removes the last two vertices

y = vertices[-1][1]

#sets the y coordinate to the last valid vertex, so a new search can be executed

else:

if changeY == True:

#the current vertex did not create a valid path so the coordinates are changed

#the search for the next vertex is started again as the conditions are different this time

y = 0

changeY = False

else:

y = vertices[-1][1]

#the search starts from the column of the previous vertex - this condition is used in the next iteration

x = vertices[-1][0]

#the x coordinate of the last vertex is used to find the next vertex in that row

recurse = False

#the function does not need to be recursed through

else:

y = 0

#the search starts from the far left cell

while vertexFound == False:

#the code is executed until a vertex is found

index = [x,y]

if y == warehouses:

#it has reached the end of the column so the search needs to be reset with different conditions

recurse = True

vertexFound = True

elif index in (skipVertex[i] for i in range(len(skipVertex))):

#if the current cell is in the array skipVertex, then the y coordinate is incremented by 1

y += 1

else:

if allocationArray[x][y] > 0:

#an allocation was made at this coordinate, so it can be used as a vertex

vertices.append([x,y])

#the new vertex is appended to the array of vertices

skipVertex.append([x,y])

#it no longer needs to be included in the search for a new vertex

vertexFound = True

#stops the loop from running again

return vertices, y, recurse, count, changeY, timer

def columnSearch(x, y, allocationArray, vertices, columns, rows, skipVertex, warehouses, factories, dummySource, recurse, count, changeY, timer, costArray, totalSupply, totalDemand):

#finds the next vertex of the path in the same column as the current vertex

vertexFound = False

#the algorithm starts at the top of every column

if recurse == True:

#if this function was just called

count += 1

#count is used to keep track of how many times this function has been called without finding a vertex

if count > warehouses:

#after the function has been run this many times, the current path is incorrect so the previous two vertices must be removed and a different path must be tried

count = 0

#set back to zero as the vertices have been changed

vertexFound = True

timer += 1

#count is used to keep track of how many times the row search has been executed without finding a vertex

vertices.pop()

vertices.pop()

x = vertices[-1][0]

#sets the x coordinate to the last valid vertex, so the next vertex can be found from this

else:

x = 0

#starts the search again from the top of the column, missing out last vertex because it didnt create a valid path

y = vertices[-1][1]

recurse = False

else:

x = 0

while vertexFound == False:

#the code is executed repeatedly until a vertex is found

index = [x,y]

#used as a comparison

if x == factories:

#the end of the column has been reached so the function needs to be executed again with the new conditions

recurse = True

vertexFound = True

#updating these Boolean variables will ensure the function is recursed

elif index in (skipVertex[i] for i in range(len(skipVertex))):

x += 1

#if you are currently at a cell that can be skipped the x index is incremented by 1

else:

if allocationArray[x][y] >0:

#if there is an allocation a vertex has been found

vertices.append([x,y])

#the vertex is added to the array of vertices

skipVertex.append([x,y])

#the vertex will now be ignored when completing the path

vertexFound = True

else:

pass

allocations = numLocationsUsed(factories, warehouses, allocationArray)

#returns the number of locations where allocations have been made

degenerate = degeneracyCheck(factories, warehouses, allocations)

#must check if new solution is degenerate

if degenerate == True:

return None, True

else:

path = pathFound(vertices)

#checks if the path is complete

vertices = verticesCheck(vertices, factories)

#checks vertices are valid

length = len(vertices)

#used as part of a condition

if path == True and len(vertices) > 2:

return vertices, False

#the path has been found and optimisation can now take place

else:

vertices, y, recurse, count, changeY, timer = rowSearch(x, y, allocationArray, vertices, columns, rows, skipVertex, factories, dummySource, recurse, count, changeY, timer, warehouses)

#calls the function that finds the next vertex

#the functions alternatively execute - column, row, column etc

path = pathFound(vertices)

#checks if the path is complete

if length >= factories and path == True :

return vertices, False

#the path has been found and optimisation can take place

else:

return columnSearch(x, y, allocationArray, vertices, columns, rows, skipVertex, warehouses, factories, dummySource, recurse, count, changeY, timer, costArray, totalSupply, totalDemand)

#if a path hasn't been found, then a new vertex needs to be found in the same column

def costChange(vertices, rawData):

#updates the costs of the current path

#the costs will alternate with positive and negative values

costs = []

#declaring a variable that will contain the costs of each vertex of a path

for i in range(len(vertices)):

costs.append(rawData[vertices[i][0]][vertices[i][1]])

#appending the cost of each vertex to the variable 'costs'

for j in range(1, len(vertices), 2):

costs[j] \*= -1

#making every other vertex a negative cost starting with the second vertex

netCost = sum(costs)

#adding all the costs in the path together

return netCost, costs

#returning information calculated

def verticesCheck(vertices, factories):

#checks that the vertices for the path are valid

fixed = False

#used as a condition for the while loop

while fixed == False:

if len(vertices) <= 2:

#if there are less than 3 vertices a path cannot be created, so the vertices are returned to be manipulated further

return vertices

else:

for p in range(1, len(vertices)):

if p >= len(vertices)-1:

#sometimes greater than due to recursion, if so the vertices are returned and assumed to be valid

return vertices

elif vertices[p-1][0] == vertices[p][0] and vertices[p][0] == vertices[p+1][0]:

#if three vertices lie on the same column, the middle one is removed and the new vertices are checked again, through recursion

vertices.pop(p)

#in-built function that removes the index 'p' from the array 'vertices'

verticesCheck(vertices, factories)

#passes the updated vertices as a parameter

fixed = True

#this stops the while loop running, once the vertices have been validated

elif vertices[p-1][1] == vertices[p][1] and vertices[p][1] == vertices[p+1][1]:

#if three vertices lie in the same row, the middle one is removed and the new vertices are checked again using recursion

vertices.pop(p)

verticesCheck(vertices, factories)

fixed = True

else:

pass

def steppingStone(rows, columns, window, factories, warehouses, allocationArray, rawData, costArray, allocations, totalSupply, totalDemand):

#function is automatically run after a method is chosen to optimise the initial solution

dummySource = True

recurse = False

changeY = False

#all used as Boolean conditions

timer = 0

count = 0

#both used as count variables to keep track of different loops

degenerate = degeneracyCheck(factories, warehouses, allocations)

#returns whether or not the solution is degenerate

if degenerate == True:

return None, None, True

#returns these values so that the initial feasible solution will be used as the final solution

else:

optimalSolution = False

#the solution is not degenerate so can be optimised further

while optimalSolution == False:

emptyCells = []

#keeps track of cells that have not had any allocations

currentLowestCost = 0

#used as a condition later on

for p in range(factories):

for q in range(warehouses):

#loops through the allocationArray

if allocationArray[p][q] == 0:

emptyCells.append([])

#appends an empty index to the empty cells array if no allocations have been made at the current coordinates

emptyCells[-1] = [p,q]

#updates the emptyCells array appended with the coordinates of the current cell

else:

pass

skipVertex = [0]

#declares an array that will be filled with coordinates

for p in range(0,len(emptyCells)):

#a path will be created for each empty cell

vertices = []

#an array declared that will have the coordinates of the current path being created

x = emptyCells[p][0]

y = emptyCells[p][1]

vertices.append([x, y])

#the starting vertex is the current empty cell

skipVertex = deepcopy(emptyCells)

#all empty cells are ignored because the path can only contain one empty cell, which is the first vertex

vertices, degenerate = columnSearch(x, y, allocationArray, vertices, columns, rows, skipVertex, warehouses, factories, dummySource, recurse, count, changeY, timer, costArray, totalSupply, totalDemand)

#calls the function that returns the path found for the current empty cell

if degenerate == True:

return None, None, True

#ensures the initial solution is used because degenerate solutions cannot be optimised

else:

pass

completeVertices = verticesCheck(vertices, factories)

#returns the path of vertices that have been validated

netCosts, costs = costChange(completeVertices, rawData)

#finds out whether or not the path can be optimised further

if netCosts < currentLowestCost:

#if the netCost is negative, the path can be optimised

currentLowestCost = netCosts

optimalVertices = deepcopy(vertices)

#a copy is used to show the difference of the original and optimised vertices

newCosts = deepcopy(costs)

#finds the path with the most negative netCost and keeps a copy of that path and it's costs

#this condition is checked at the end of every iteration, ie, every time a new path has been created

finalAllocations = deepcopy(allocationArray)

#the 'finalAllocations' array will be changed and optimised so a copy of the solution is kept

if currentLowestCost < 0:

#if the path can be optimised

lowestAllocation = inf

#will be used as a comparison value

for k in range(1, len(optimalVertices), 2):

#ignores the empty cell since no allocations will be made there

#finds the smallest allocation for the vertices with a negative cost

currentAllocation = finalAllocations[optimalVertices[k][0]][optimalVertices[k][1]]

#finds the allocation made at the current vertex

if currentAllocation <= lowestAllocation:

#finds the smallest allocation for the negative vertices

lowestAllocation = currentAllocation

xIndex = optimalVertices[k][0]

yIndex = optimalVertices[k][1]

#saves the coordinates of the vertex with the lowest allocation

minimumAllocation = finalAllocations[xIndex][yIndex]

#updates the value of the lowest allocation

for j in range(len(optimalVertices)):

x = optimalVertices[j][0]

y = optimalVertices[j][1]

#declares the coordinates of the current vertex as 'x' and 'y'

if newCosts[j] > 0:

#if the cost of the current vertex is positive we add the minimum allocation

finalAllocations[x][y] += minimumAllocation

else:

finalAllocations[x][y] -= minimumAllocation

#if the cost is negative, we subtract the minimum allocation

allocationArray = finalAllocations

#the allocationArray identifier is used when the code executes on the next iteration

#the optimisation algorithm is then used on the updated array

else:

optimalSolution = True

cost = costCalculate(factories, warehouses, costArray, finalAllocations)

#the cost of the optimised array is calculated

return finalAllocations, cost, False

#the optimised data is returned to be displayed on the results page

def helpMainPage():

#help information for the main page

tk.messagebox.showinfo("Help", "The number of Factories and Warehouses are the total number of places you are planning to move items from/ to. \n\nThe demand and supply are the total quantities you have in all the factories and all the warehouses.\n\nThe execution button will show you a window where you can enter the rest of your data and select a method.")

#one piece of information the user can read if the help button for this window is selected

def helpMatrix():

#help information for the input matrix

tk.messagebox.showinfo("Help", "The cells labelled with 'W' and 'F' represent your factories and warehouses. The cells at the end of each row/ column are for the total supply/ demand for that factory/warehouse. The other cells represent the transportation cost for the factory to warehouse at that index.")

#more information the user can access if needed

def helpMethods():

#help information for the available methods

tk.messagebox.showinfo("Help", "There are three different methods to produce a solution. To find the most efficient for your data compare the results of each.\n\n A results window will open to show the optimal solution to the method you click on.")

def mainMenu():

#declares all variables and information needed to create the interface

warehouses = tk.StringVar()

factories = tk.StringVar()

totalSupply = tk.StringVar()

totalDemand = tk.StringVar()

#initialising variables to be used on first window

img = tk.PhotoImage(file = "lorry.png")

#original size is larger than necessary

smallImg = img.subsample(7,7)

#makes image 1/3 of the size by taking every third pixel from x and y

tk.Label(root, image = smallImg).grid(row = 0, column = 1)

#places image onto the window

ttk.Label(text = "No. warehouses").grid(row = 1, column =0)

warehouse\_box = ttk.Entry(root, textvariable = warehouses, width = 4)

#creates a label with a box for the user to enter the stated information

warehouse\_box.grid(row = 1, column =1)

ttk.Label(text = "No. factories").grid(row = 1, column =2)

factory\_box = ttk.Entry(root, textvariable = factories, width = 4)

#matches each entry box input to a different variable to be used later on

factory\_box.grid(row = 1, column =3)

ttk.Label(text = "Total Supply").grid(row = 2, column =0)

factory\_box = ttk.Entry(root, textvariable = totalSupply, width = 4)

factory\_box.grid(row = 2, column =1)

ttk.Label(text = "Total Demand").grid(row = 2, column =2)

factory\_box = ttk.Entry(root, textvariable = totalDemand, width = 4)

factory\_box.grid(row = 2, column =3)

ttk.Button(root, text = "execution", command = lambda: matrixTable(factories, warehouses, totalSupply, totalDemand)).grid(row = 3, column =1)

#creates a button the user can click on to go to the next window

ttk.Button(root, text = "help", command = lambda: helpMainPage()).grid(row = 3, column = 2)

#outputs extra information to the user if needed

root.mainloop()

#enables us to run the program

#creates an infinite loop waiting for the user to give it a task and will carry on until the program (interface) is closed

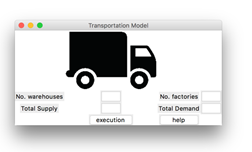
mainMenu() #calls the first function, starting the program

# Testing

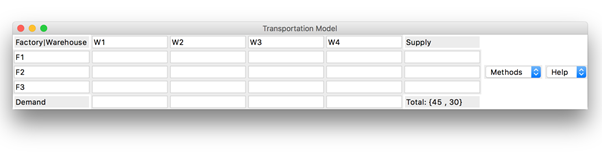
Numerous tests were carried out, of which the core tests are for validation. Other areas were also tested including: tests to ensure the code executes correctly, the interface produces the correct information and any widgets used are executed correctly.

There are references to ‘first window’, ‘second window’, ‘third window’ and ‘original solution window’. An example of what they look like are shown below:

## First Window:



## Second Window:



## Third Window:



## Test evidence

The tests are all explained below and there is a video showing the outcome of each test at the following link:

<https://youtu.be/_lMgIGLGIek>

## Test scenarios

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Area** | **Purpose** | **Test No.** | **Test Data** | **Data Type** | **Expected Result** | **Pass / Fail** |
| Program Code –  Checking to see if the functions produce the correct output | To ensure the three main algorithms produce the correct initial feasible solution and the costCalculate and other associated functions produce the correct results | 1.1.1 | Using the balanced dry run data for each method  Method = North West Corner | Normal | The initial feasible solution will be shown on the original solution window and should match the solution of the dry run data. | pass |
| 1.1.2 | Method = Least Cost | Normal | The initial feasible solution will be shown on the original solution window and should match the solution of the dry run data. | pass |
| 1.1.3 | Method = Vogel’s Approximation | Normal | The initial feasible solution will be shown on the original solution window and should match the solution of the dry run data. | pass |
| 1.1.4 | Using the unbalanced dry run data for each method  Method = North West Corner | Boundary | The initial feasible solution will be shown on the original solution window and should match the solution of the unbalanced dry run data. | pass |
| 1.1.5 | Method = Least Cost | Boundary | The optimised solution will be shown on the results page and should match the results of the dry run unbalanced optimised data. | pass |
|  |  | 1.1.6 | Method = Vogel’s Approximation | Boundary | The initial feasible solution will be shown on the original solution window and should match the solution of the dry run unbalanced data. | pass |
| To ensure the optimisation algorithm finds the solution with the lowest cost and checks the steppingStone function works alongside the associated functions | 1.2.1 | Using the balanced dry run data  Method = Vogel’s Approximation | Normal | The optimised solution will be shown on the results page and should match the results of the dry run optimised data. | pass |
| 1.2.2 | Using a degenerate solution  Method = Least Cost | Erroneous | The optimised solution will be shown on the results page and should match the results of the dry run optimised data. | pass |
| 1.2.3 | Using the unbalanced dry run data  Method = North West Corner | Boundary | The optimised solution will be shown on the results page and should match the results of the dry run unbalanced optimised data. | pass |
| 1.2.4 | Method = Least Cost | Boundary | The optimised solution will be shown on the results page and should match the results of the dry run unbalanced optimised data. | pass |
| 1.2.5 | Using an unbalanced degenerate solution  Method = Vogel’s Approximation | Erroneous | The optimised solution will be shown on the results page and should match the results of the dry run unbalanced optimised data. | pass |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | To ensure data can be entered into the first window and the matrixTable and collectData functions work correctly | 1.3 | Enter random data | Normal | The first window opens and data can be entered into the boxes. The second window is then produced from the click of the execution button | pass |
| To ensure data can be entered into the second window, the drop down menu works and the menuOptions function works correctly | 1.3.2 | Enter random data into the table and click one of the method menu options | Normal | The entry boxes can be typed into and the menu drops down to show the different methods that can be chosen | pass |
| User Interface - Checking the validation of user inputs on the first window and checking the validateMain-Window function executes correctly  -  Checking the classes produce a suitable interface | To ensure normal random data is accepted by the program. | 2.1 | Factories = 3  Warehouses = 4  Total supply = 45  Total demand = 80 | Normal | The second window will open with the matrix having the correct dimensions and correct labels | pass |
| To ensure you can use the upper and lower limits of the valid inputs. | 2.2 | Factories = 2  Warehouses = 6  Total supply = 2  Total demand = 6 | Boundary | The second window will open with the matrix having the correct dimensions and correct labels | pass |
| To ensure the user cannot enter values higher than the upper limit | 2.3 | Factories = 7  Warehouses = 2  Total supply = 40  Total demand = 55 | Erroneous | An error message will pop up stating: ‘The maximum number of factories is 6, maximum number of warehouses is 6.’ | pass |
| To ensure invalid data is not accepted by the program. | 2.4 | Factories = 2  Warehouses = 2  Total supply = 0  Total demand = 55 | Erroneous | An error message will pop up stating: ‘You must have at least 2 supply and 2 demand’ | pass |
| To ensure if there are several errors with the data, that they are all shown to the user. | 2.5.1 | Factories = 0  Warehouses = 0  Total supply = 0  Total demand = 0 | Erroneous | An error message will pop up stating: ‘There must be at least 2 factories and 2 warehouses.’ | pass |
| To ensure if there are several errors with the data, that they are all shown to the user. | 2.5.2 | Factories = 2  Warehouses = 2  Total supply = 0  Total demand = 0 | Erroneous | An error message will pop up stating: ‘You must have at least 2 supply and 2 demand’ | pass |
| 2.5.3 | Factories = 2  Warehouses = 2  Total supply = 2  Total demand = 2 | Normal | The second window will open with the matrix having the correct dimensions and correct labels | pass |
| To ensure the user cannot enter values that have a simple solution | 2.6 | Factories = 1  Warehouses = 1  Total supply = 30  Total demand = 67 | Erroneous | An error message will pop up stating: ‘There must be at least 2 factories and 2 warehouses. If there is only 1 factory or 1 warehouse, then all items are sent from/ delivered to that factory/ warehouse.’ | pass |
| To ensure letters are not accepted as an input | 2.7 | Factories = 3  Warehouses = p6  Total supply = 40  Total demand = m | Erroneous | An error message will pop up stating: ‘You must enter integer numbers.’ | pass |
| To ensure decimal values are not accepted as an input | 2.8 | Factories = 3.5  Warehouses = 2  Total supply = 40  Total demand = 50 | Erroneous | An error message will pop up stating: ‘You must enter integer numbers.’ | pass |
| To ensure negative values are not accepted as an input | 2.9 | Factories = 3  Warehouses = 2  Total supply = -5  Total demand = 50 | Erroneous | An error message will pop up stating: ‘You cannot have negative values.’ | pass |
| To ensure special characters are not accepted as an input | 2.10 | Factories = 3  Warehouses = \*  Total supply = ?  Total demand = 50 | Erroneous | An error message will pop up stating: ‘You must enter integer numbers.’ | pass |
| User interface – Checking the buttons work on the first window | To ensure the buttons execute the correct commands and the helpMainPage function executes correctly | 3.1 | Pressing the help button | Normal | The help information box should pop up | pass |
| To ensure the buttons execute the correct commands and the helpMainPage function executes correctly  To ensure the input matrix accepts valid inputs | 3.2.1 | Factories = 3  Warehouses = 2  Total supply = 40  Total demand = 5  Pressing the execution button | Normal | The second window should pop up | pass |
| 3.2.2 | Pressing the execution button with no input | Erroneous | An error message should pop up stating: ‘You must enter integer numbers.’ | pass |
| 4.1 | Factories = 3  Warehouses = 4  Total supply = 30  Total demand = 67  The costs and individual supply/ demands used are the same as the data used in the dry run data.  Any method can be clicked to validate the data, for this test I will use North West Corner. | Normal | The third window should pop up, with no error messages. | pass |
| User Interface – Checking the validation of user inputs on the second window and checking the validateInputs function produce the correct result | To ensure it is possible to change the names if the user chooses | 4.2 | The factory and warehouse names will be replaced with names | Normal | The third window should pop up, with no error messages. | pass |
| To ensure the user can input the lower limit values and the program accept it. There are no upper limits for the cost | 4.3 | Some of the costs will be replaced with zeros | Boundary | The third window should pop up, with no error messages. | pass |
| To ensure if the inputs are invalid, an appropriate error is shown | 4.4.1 | All costs will be replaced with zeros | Erroneous | An error message will pop up stating: ‘You must enter several costs above zero or all routes will be free, so there is no optimum solution.’ | pass |
| To ensure if the inputs are invalid, an appropriate error is shown  To ensure if there are multiple errors with the data, they are all shown to the user | 4.4.2 | All costs will be replaced with the same number | Erroneous | An error message will pop up stating: ‘When all values are the same, there is no optimal solution, all routes will have the same outcome.’ | pass |
| 4.5.1 | Some of the cost cells will be left empty and some replaced with a letter or a letter and a number | Erroneous | An error message will pop up stating: ‘You must enter an integer value into every empty cell’ | pass |
| To ensure if there are multiple errors with the data, they are all shown to the user  To ensure if the user has incorrectly added up their supply and demand, an appropriate error message will be shown | 4.5.2 | The empty cells will be replaced with integers | Erroneous | An error message will pop up stating: ‘You must only enter whole numbers for the costs, supply and demand.’ | pass |
| 4.5.3 | All erroneous data will be replaced with valid inputs | Normal | The third window should pop up, with no error messages. | pass |
| 4.6.1 | The supply and demand will not add up to the total values entered in the first window | Erroneous | An error message will pop up stating: ‘Your supplies do not add up to the total you entered.’ | pass |
| To ensure if the user has incorrectly added up their supply and demand, an appropriate error message will be shown  To ensure decimal values are not accepted as an input | 4.6.2 | The supply column will be changed so it adds up to the correct amount | Erroneous | An error message will pop up stating:  ‘Your demands do not add up to the total you entered.’ | pass |
| 4.7 | Some of the costs will be changed to decimal values | Erroneous | An error message will pop up stating: ‘You must only enter whole numbers for the costs.’ | pass |
| To ensure negative numbers are not accepted as an input | 4.8 | Some of the costs will be changed to negative values | Erroneous | An error message will pop up stating: ‘You must only enter numbers above zero.’ | pass |
| To ensure special characters are not accepted as an input | 4.9 | Some of the entries will be changed to special characters | Erroneous | An error message will pop up stating: ‘You must only enter whole numbers for the costs, supply and demand.’ | pass |
|  | To ensure if the user removes a row/ column label, they replace it with a new string | 4.10 | One of the labels will be left blank | Erroneous | An error message will pop up stating: ‘You must enter a value into every empty cell.’ | pass |
| User Interface – Checking the menu options and help options work on the second window | To ensure the method choices all produce a results page | 5.1.1 | Factories = 3  Warehouses = 4  Total supply = 30  Total demand = 30  The costs and individual supply/ demands used are the same as the data used in the dry run data.  Method = North West Corner | Normal | The third window should pop up, with no error messages | pass |
| 5.1.2 | Method = Least Cost | Normal | The third window should pop up, with no error messages | pass |
| 5.1.3 | Using a degenerate solution  Method = Vogel’s Approximation | Erroneous | The third window should pop up, with a line telling the user it is a degenerate solution | pass |
| To ensure the help options all produce an information box | 5.2.1 | Entering random data and then clicking the matrix help option | Normal | An information box explaining the different cells of the input matrix should pop up | pass |
| 5.2.2 | Entering random data and then clicking on the methods help option | Normal | An information box explaining what the methods are should pop up | pass |
| User Interface – Checking the results page produces the correct information | To ensure the network graph has the correct number of nodes and the correct network links | 6.1.1 | Factories = 3  Warehouses = 4  Total supply = 30  Total demand = 30  The connections will be compared to the dry run solution  Any method can be clicked to validate the data, for this test I will use Least Cost. | Normal | The network graph will have 7 nodes in total, 3 labelled with ‘F’ and 4 labelled with ‘W’ | pass |
| 6.1.2 | Factories = 3  Warehouses = 4  Total supply = 30  Total demand = 67  Using the unbalanced solution to check dummy sources are not shown  Method = Least Cost | Boundary | The network graph will have 7 nodes in total, 3 labelled with ‘F’ and 4 labelled with ‘W’. The dummy node will not be displayed and neither will any connections to it | pass |
| To ensure the results table shows the correct data | 6.2 | Factories = 3  Warehouses = 4  Total supply = 30  Total demand = 30  The connections will be compared to the dry run solution  Any method can be clicked to validate the data, for this test I will use Least Cost. | Normal | The results table will show the number of allocations made at each node. It will match the results of the dry run data | pass |
| To ensure the correct information is provided to the user | 6.3 | Checking the text written on the page is clear and understandable to the user  Any method can be clicked to validate the data, for this test I will use Least Cost. | Normal | The following should be stated on the third window: the cost of the solution, the name of the solution and a sentence stating what the network graph is. | pass |
| 6.4 | Using a degenerate solution  Method = North West Corner | Erroneous | The following should be stated on the third window: the cost of the solution, the name of the solution, the solution being degenerate and a sentence stating what the network graph is. | pass |
| To ensure each results page is identified by the correct tittle | 6.5.1 | Method = North West Corner | Normal | The results page should open with the title ‘North West Corner’ | pass |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 6.5.2 | Method = Least Cost | Normal | The results page should open with the title ‘Least Cost’ | pass |
| 6.5.3 | Method = Vogel’s Approximation | Normal | The results page should open with the title ‘Vogel’s Approximation’ | pass |
| To ensure the user can view the original solution if wanted and checks the originalSolution function works | 6.6.1 | Original solution is wanted | Normal | The third window and another window titled ‘Original Solution’ will open and the original solution will be displayed on it, with the network graph for the original solution | pass |
| 6.6.2 | Original solution is wanted and is degenerate | Erroneous | An information box will pop up stating: ‘The solution is degenerate so the original solution is the same as the optimum solution.’ When this is closed, the results page should open | pass |
| 6.6.3 | Original solution is not wanted | Normal | The results page should open | pass |
| To ensure the user can open multiple results windows at once | 6.7 | All methods are used | Normal | The results page for each method should pop up with no error messages | pass |

All the tests I carried out were successful, showing the robustness of my code and the efficiency of error checking within my code.

# Evaluation

## Comparison of the project against the objectives:

1: Create a Tkinter interface

1.1.1: A menu page was created that allowed the user to enter data for all stated variables. The user was also able to click a button labelled ‘execution’, which took them to the next window. There was also a ‘help’ button added to the interface, which gave the user information about the data to be entered on this window. Therefore, objective 1.1.1 was completed.

1.1.2: An entry page was created with a matrix that allowed users to enter their data. This matrix had labels showing what each row and column represented. There was also a drop-down menu created, so the user could choose which method they wanted results for. This menu could be used as many times as the user wanted, so all three methods could be displayed at one time. Also, a ‘help’ drop-down menu was created giving the user the choice of accessing information about the matrix and what the methods are. Therefore, objective 1.1.2 was completed.

1.1.3: A results page was created, which showed the user a network graph of all the allocations made, as well as, a table showing the quantity of allocations made for each factory to warehouse route. The user is also given the option to view the original solution, that has not been optimised. This window shows the network graph and table of allocations for the original solution. The cost for each solution is also shown to the user. Therefore, objective 1.1.3 was completed.

1.2.1: The built-in Tkinter widgets were used to create all the menus and buttons on the interface, this saved time when completing the code. They all used the themed Tkinter widgets, so the user was familiar with the look of them. Therefore, objective 1.2.1 was completed.

1.2.2: For each button and menu, a meaningful title was given to them so the user could understand what each widget does. The identifiers of the variables in the code also used meaningful names. Therefore, objective 1.2.2 was completed.

1.2.3: Every button and menu option had a different function to execute. The correct functions were called for each button/ menu option. Therefore, objective 1.2.3 was completed.

2: Calculate initial costs for every selected method

2.1: The ‘collectData’ function is called once the user has chosen a method. This ensures all user data is collected and stored before any manipulation can take place. Therefore, objective 2.1 was completed.

2.2: Since all data has already been collected and stored before the selected method is executed, all data that Is used, is used for the correct purpose. This ensures the correct solution is produced. Therefore, objective 2.2 was completed.

2.3: The ‘degeneracyCheck’ function is called after every initial solution has been found and whilst an optimised solution is being produced. This ensures the user is told if a solution is degenerate and means the optimisation algorithm will not run for longer than necessary. Therefore, objective 2.3 was completed.

3: Optimise initial solutions

3.1: Once an initial solution has been found, the Stepping Stone algorithm is executed to see if the solution can be optimised. Therefore, objective 3.1 was completed.

3.2: After the final solution has been found, the results page is created using the ‘analysisClass’ class. This passes the final solution and its corresponding data through several methods, to create the network graph and table of allocations to be displayed don the results page. Therefore, objective 3.2 was completed.

## User feedback:

Student 1 found the interface to be simple to understand and finds the help buttons very useful. Also, the way in which I implemented the three methods within the interface allowed student 1 to understand what they do. This enabled student 1 to find the cheapest delivery scheme for her items ordered online.

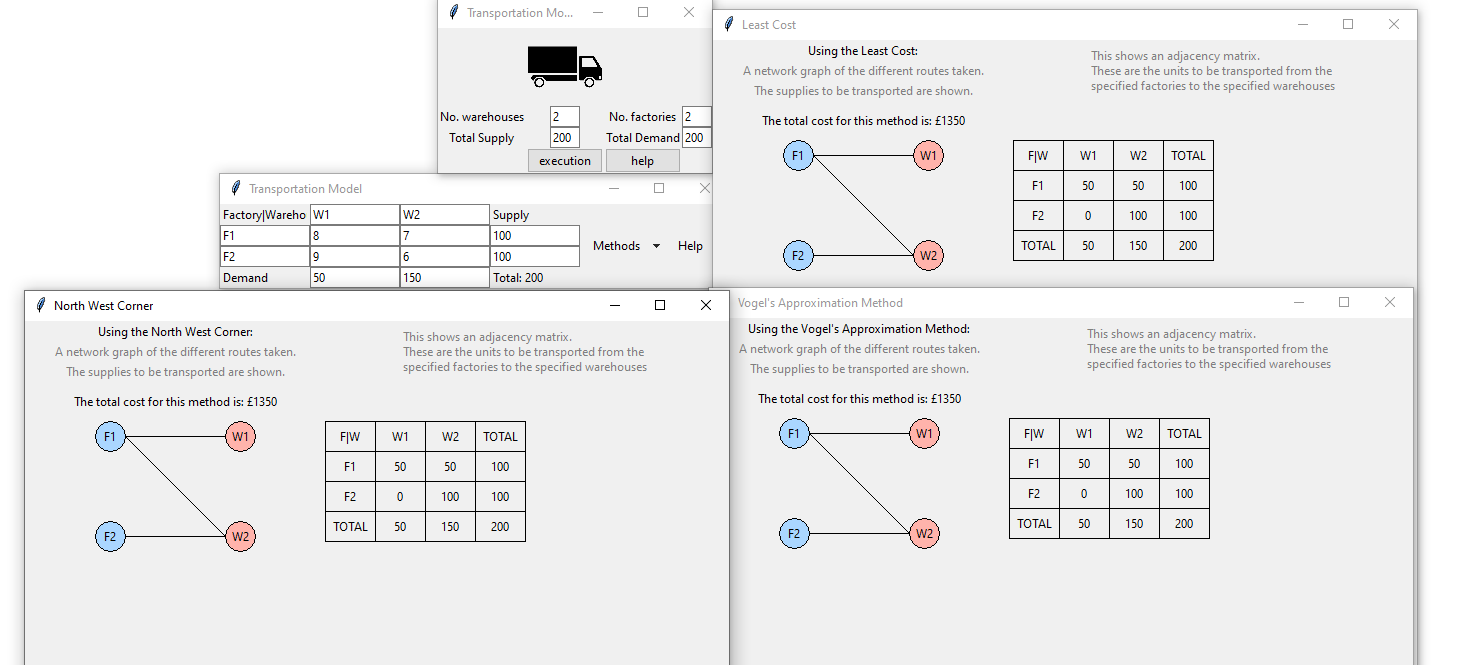
Student 2 thought the network graph and table of allocations were structured well and helped them to fully understand the difference between each method. They found it quick and easy to find the method with the lowest cost, due to the easy comparisons that can be made. This allowed student 2 to grasp a better understanding of what the transportation model is and the way in which different costs can be found.

## Possible improvements:

After reviewing the program and interface, a few possible improvements that could be made, if more time was given, are:

* Including the full factory and warehouse names as the labels for the table of allocations.
* Being able to save and load matrix data and results from/ to a file.
* Using the optimisation algorithm on degenerate solutions
* Creating unit tests to confirm the algorithm work and enabling independent tests of the algorithms and the interface

## Outcome of the investigation

After using my program to run numerous tests for each method, I have established that balanced solutions, with an equal number of factories and warehouses, are optimised to produce the same cost. An example of this is:

As well as this, I found that almost all unbalanced solutions were degenerate, so the optimisation algorithm could not be executed on them. However, for the few unbalanced solutions that were not degenerate, I found that, after optimisation, the three methods did not produce the same cost. There was no method that produced a lower cost more consistently than the other two methods.

The three algorithms are suited for implementation on a computer and don’t take much time to execute. As a consequence of this, it encourages the user to experiment to see what the consequences of additional factories or warehouses may be, allowing a company to see how the expansion of their company could work without having to spend money.

A different approach I could have taken to the investigation was to use a different number of methods. If I were to include less methods, there would not be as wide a range of data to compare costs between. If I were to include more methods, the program code would take longer to write and implement; given the time limits of this project, implementing more methods would be inefficient. Therefore, by using the three methods I coded, a sufficient range of data could be produced to draw an accurate conclusion.

The user feedback was all positive, suggesting a successful program that accomplished all user requirements. The objectives were also all met, showing that even though there are improvements that could be made, the program is efficient. I believe this was an effective investigation, and given more time, more conclusions could be drawn from the data allowing me to draw more conclusions about the optimisation of the three algorithms.

1. Found on <https://www.fingent.com/blog/how-can-logistics-and-transportation-companies-use-digital> [↑](#footnote-ref-1)
2. Found on <https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/651257/road-freight-stats-april-2016-to-march-2017.pdf> [↑](#footnote-ref-2)