**Overall design**

To make the project easier to handle, it will be split up into several tasks. This way, it reduces the chance of mistakes that may be presented later on in development.

Use Story 1: User can set number of rows and columns for the problem

**User Task 1:**

User can set the numbers of rows and columns for the program, so that the required grid is created for the user.

Create a GUI for the user to use to decide the rows and columns needed for the problem.

This is a simple task that only involves with the design of the first GUI the user will arrive at. On the GUI will include areas that the user can type and multiple buttons including:

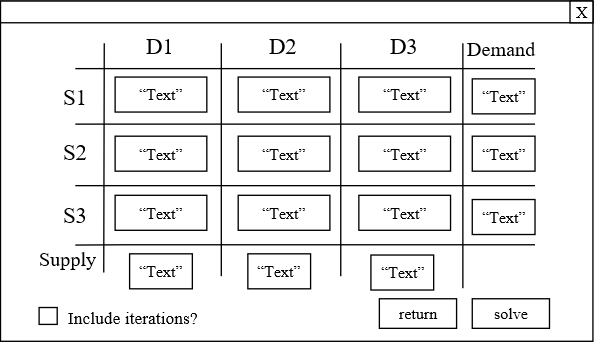
* Create – this will open another GUI that will let the user enter values that are needed to solve the problem.
* Help – this will open a message box explaining what is happening on the window it’s currently opened at.
* Random – this is the optional task that creates an another GUI with random values, therefore, creating a transportation problem.

After the user clicks on the “create” button, the values entered will be checked to see if they are numbers and in between the allowed range of numbers. After that, the values will be used to create the grid window.

Flowchart of the basic operations.

**Task 2:**

Grid window

Makes the program create a custom grid window depending on the rows and columns, and allow the user to enter the values necessary for the problem.

Checks and Solves the transportation problem

Returns back to the main menu

Using the row and columns, the text boxes will be placed in a grid shape, making it easier for the user to recognize where to put the values. Also on the grid window will be tick boxes that decides if the user wants to have some iterations in the excel file or not. The reason why this is done is that if the user does want it to be exported, it will be quicker and easier to post each iterations on the excel file before starting the next one, and therefore the program does not need to be used again.

There will also be a button that checks if the values are integers and are suitable for the problem created. After that, the values will be used to calculate the solution.

**Task 3:**

Using a series of methods and algorithms, the program will be able to use the values from the grid window and produce an optimal solution.

To make this as organised as possible, a class will be used to group individual data into objects. Each object will be a cell that is in the table. It will hold multiple information like the delivery costs, how much stock it will deliver and the product of the combined values. They will also have coordinates so that the pointer can call the right cell and amend it if needed. Arrays will also be used to hold data that is “outside” of the cells, like the total supply, total demand and shadow cost of each row and column.

To make sure that the information can be used without causing an error, each value is checked weather if it is an integer or not.

Another note to state is that the all of the methods will be done in a separate file. The reason why is:

* The only times a GUI is needed is when the user needs to enter information and to see the result. So having a GUI for the methods is unnecessary and time-consuming.
* Hence, if the methods were done in a different file, a constant set of data can be used instead of manually inputting information, saving time on testing and evaluating.
* It is also easier to find errors in the code as it is purely on the methods.
* All of the methods and algorithms are “maths-heavy”, which means that having a clean file to start with will make it easier to code and understand where to look at if there is a mistake.



**Task 4:**

To be able to code the north-west corner method, creating an initial solution for the problem.

before the method is used, if the problem is unbalanced (more stock than needed) then a dummy column is needed. This means extra cells are created.

The method will go from the north-west corner of the grid and uses as much stock as possible to give to the customers, then move is a stair like shape to fill the requests. For this to work, some kind of pointer is needed to get info on stock amount, stock needed and how much stock is being transferred to that customer, hence the reason why the cells need coordinates so that the right information is gathered.

One problem is how to tell the pointer when and where to move after doing a cell. One idea is that arrays in an organised way are created. This means that once the pointer is done with the first cell, it checks the arrays of the stock amount and stock needed, and move depending if one value is full or both are full.

**Task 5:**

After the north-west corner method, the program needs to find the shadow costs of the problem.

As this is “outside” of the grid, an array will hold the necessary information for this task and not in a cell object. The shadow cost is calculated by starting from the north-west corner, make the source zero, and calculate the other by taking away the shadow cost of that row from the delivery cell cost. Once all of the non-empty cells have been done, then it starts on the next row, calculates the row shadow cost be taking away the column shadow cost from the cell. The whole process repeats.

Again, some sort of pointer is needed to know what cell is being used. The advantage is that the path of the pointer is linear. For example, once all of the non-empty cells have been done, it can go straight to the first cell of the next row and check if there is stock being delivered in that stock.

**Task 6:**

To find the improvement indices to check if the solution is optimal and if not, find the entering cells.

The improvement indices are calculated with each empty cell. This means that the improvement cells are stored in the cell object itself. This is calculated by taking away the row and column shadow cost from the empty cell delivery cost. If all of the improvement indices from the empty cells are positive values, then a optimal solution has been found. If there is a negative value, then the current solution is not optimal. The lowest negative improvement indices will be the place of the entering cell.

As the improvement indices are stored in cells, it makes the process of calculating the value somewhat easier, as the coordinates of the cell can tell which shadow cost value from the row and column is needed for that cell. Another pointer is used to find the empty cells, unless the pointer used for the shadow costs is effectively used so that it organises which cell is empty or non-empty.

**task 7:**

using the entering cell as a guidance, find which cells are affected by it and make the required calculations using the stepping-stone method.

The rules to find which cells are affected by the entering cells are:

1. There can only be one increasing and one decreasing cell in each row and column.
2. The affected cells can only be the entering and non-empty cells.

Once all of the affected cells have been found, the entering cell number is the smallest number of the cells that are decreasing. For example: if one cell is 5 - α and another is 10 - α, then α can only be 5 as its impossible to get negative stock, which means then that the first cell has 5 stock and the other has 0 stock.

This is the most difficult task in the project. One solution to do is to create another “virtual” grid that only tells which cells are non-empty, the entering cell, how many cells are affected in that row or column, which cells out of the affected is increasing and decreasing and the value of stock that the cell holds. The grid is checked every time to see if it has followed the rules stated above. If not, the numbers stating how many cells are affected in that row and column is checked. If there is a “1”, the pointer will find an unaffected cell in that row or column and will make it increasing or decreasing, depending of what is already in that row or column.

But once the grid has passed the rules, the value for the entering cell is calculated. This Is done by filtering which cells are decreasing, then entering cell value is the smallest value from them cells. next, A pointer that will be directly connected to both grids by coordinates will tell each non-empty cells what to do depending if it is affected and increasing or decreasing by the entering cell value. In the end, only one cell should have zero stock and another have stock.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 0 | 2 |
| 0 | \* |  |  |
| 2 | -α | \* | α |
| 1 | \* | \* | -α |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 2 | 0 | 2 |
| 0 | \* |  |  |
| 2 | -α | \* | α |
| 2 | +α | \* | -α |

complete grid – all rows and columns have two affected cells

Incomplete grid – one row and one column only have one affected cell.

|  |  |
| --- | --- |
| key: |  |
| \* | non -empty cell |
| α | entering cell |
| -+α | affected cell |
| 0,1,2 | number of affected cells in a row/column |

**Task 8:**

After the optimal solution has been found, a window will appear showing the optimal solution in a grid form.

if the problem was unbalanced, then an extra column is needed to show this. There will also be a button that allows the user to read the result in text form. For example: A will deliver 200 stock to X, B will deliver 100 stock to Z and so on. The total cost will also be calculated and shows the difference between the initial solution and the final solution cost. There will also be buttons that allows the user to export the solution and with its iterations (if the user want it from the grid window option) into excel.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| problem: |  |  |  |  |  |  |  |
|  | X | Y | Z | supply |  |  |  |
| A | 40 | 47 | 80 | 100 |  |  |  |
| B | 72 | 36 | 58 | 200 |  |  |  |
| C | 24 | 61 | 71 | 300 |  |  |  |
| demand | 200 | 200 | 200 |  |  |  |  |
|  |  |  |  |  |  |  |  |
| initial solution: | |  |  |  |  |  |  |
|  | X | Y | Z | supply |  |  |  |
| A | 100 |  |  | 100 |  | total cost initially: | |
| B | 100 | 100 |  | 200 |  | 35100 | |
| C |  | 100 | 200 | 300 |  |  |  |
| demand | 200 | 200 | 200 |  |  |  |  |
|  |  |  |  |  |  |  |  |
| final solution: | |  |  |  |  |  |  |
|  | X | Y | Z | supply |  | final total cost: | |
| A |  |  | 100 | 100 |  | 27100 | |
| B |  | 200 |  | 200 |  |  |  |
| C | 200 |  | 100 | 300 |  |  |  |
|  | 200 | 200 | 200 |  |  |  |  |

This is a view of what the file could look like without the iterations.