Coded and edited by: Yasser Almohammad. 2017

Contents

Figure List

Table list

Problem Statement

Shelf Space Allocation Problem is the problem of finding the optimal solution (or a good solution) for arranging products (or objects) on shelves in a store/warehouse to maximize profit, the problem considers different shelves with different priorities based on customer experience studies or based on certain criteria defined by the warehouse manager, while the products have different dimensions and different profit margin.

The problem can be described in a complicated context, with many demands and aggregations, and several rules, and can also be simplified and the solution is generalized for more complicated rules.

Our solution will solve this problem and present planograms in motion to show the change of the solution and the improvements while still satisfying the criteria of fitting all the objects within the minimum and maximum limits, the solution at hand can be generalized for more complicated scenarios where similar products can be stacked on top of each other, and have a certain depth count in each allocation.

The problem is a large scale optimization NP Hard problem, the solution here will present different approaches in the implementation for the same algorithm using LNS Largest Neighborhood Heuristics Search that shows how some are faster than others and all yield an improved arrangement from scratch given the products/shelves/priorities data set.

LNS Large neighborhood search:

Using LNS Large neighborhood Search to SSAP.

Heuristics based on *large neighborhood search* have been recently tested and showed great results in solving np hard problems and various large scale optimization problems such as transportation and scheduling problems, including vehicle routing problems, and the Travelling Sales Man Problem. Large neighborhood search methods explore a complex neighborhood by use of heuristics. Using large neighborhoods makes it possible to find better candidate solutions with each iteration thus traversing a more promising search path reducing the solution space that the problem is looking in; tell we find an optimal solution or stop using a specific criteria like time and return the resulting solution.

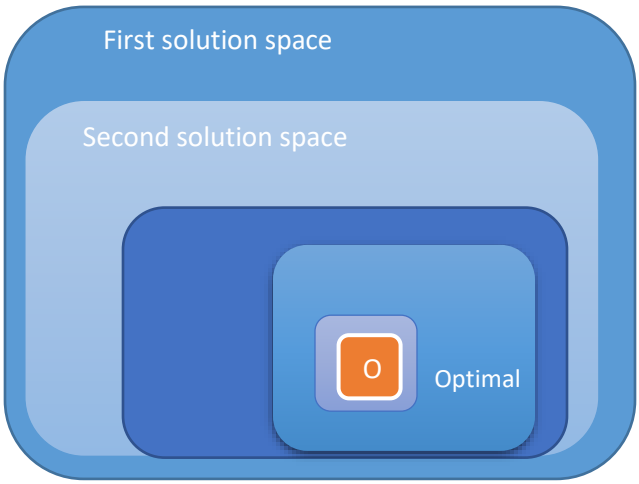


Figure 1: LNS Solution space down to the optimal solution

In the LNS the neighborhood is defined implicitly by a *destroy* and a *repair* method (try-error-destroy-repair-try…) . A destroy method destructs part of the current solution whereas a repair method rebuilds the destroyed Solution.
The destroy method typically contains an element of stochasticity such that different parts of the solution are destroyed in every invocation of the method. The neighborhood $N(x)$ of a solution x is then defined as the set of solutions that can be reached by first applying the destroy method and then the repair method.
A very simple destroy method would select the airplanes landing time at random. A repair method could rebuild the solution by inserting removed landing times tested in a previous iteration, using a greedy heuristic.
LNS heuristic typically alternates between an infeasible solution and a feasible solution: the destroy operation creates an infeasible solution which is brought back into feasible form by the repair heuristic. Alternately the destroy and repair operations can be viewed as fix/optimize operations: the *fix* method (corresponding to the destroy method) fixes part of the solution at its current value while the rest remains free, the *optimize* method (corresponding to the repair method) attempts to improve the current solution while respecting the fixed values. Such an interpretation of the heuristic may be more natural if the repair method is implemented using MIP Mixed Integer Problem or constraint programming solvers.
In our SSAP for instance we fix the current the product allocation on one shelf while the remaining values are randomly generated in search for a better solution, same applied for other variables.
Local search techniques are very effective to solve hard optimization problems. Most of them are, by nature, incomplete. In the context of constraint programming for optimization problems, The basic idea is to iteratively relax a part of the problem, then to use constraint programming to evaluate and bound the new solution using fix and optimize principle mentioned earlier.

LNS is a two-phase algorithm which partially relaxes a given solution and repairs it. Given a solution as input, the relaxation phase builds a partial solution (or neighborhood) by choosing a set of variables to reset to their initial domain; The remaining ones are assigned to their value in the solution. Even though there are various ways to repair the partial solution, the focus is on the technique in which Constraint Programming is used to bound the objective variable and to assign a value to variables not yet instantiated. These two phases are repeated until the search stops (optimality proven or limit reached of time or any other criteria defined by the developer).
All LNS methods are problem dependent, so several approaches have been proposed by researchers to generalize the LNS and make problem independent, thus the following methods have been proposed: Propagation and explanation based methods.
A simplified general algorithm for the LNS method can be viewed as:
Table 1: LNS Search method (Found in [1])

```
Algorithm :Large Neighborhood Search
Requirement: an initial solution S
procedure LNS
        while Optimal solution not found and a stop criterion is not encountered do
                relax(S)
                S'=   findSolution() .  The current partial solution is then repaired in order to
improve
```

```
                    the current solution
                if S' != NULL then . An improving solution has been found
                    S = S'
                end if
        end while
end procedure
```

PLNS:

One drawback of LNS is that the relaxation process is quite often problem dependent. Some works have been dedicated to the selection of variables to relax through general concept not related to the class of the problem treated . However, in conjunction with CP (Constraint Programming), one generic approach, namely Propagation-Guided LNS, has been shown to be very competitive with dedicated ones. It must, in a way, automatically detect the problem structure in order to be efficient.

In our usage of the solver the method is called using the following method with signature and parameters

public static void pglns(Solver solver, IntVar[] vars, int fgmtSize, int listSize, int level, long seed, ACounter frcounter)

Which Creates a PGLNS based LNS method for the Choco solver, where fgmtSize - fragment size (evaluated against log value), listSize - size of the list, level - the number of tries for each size of fragment, seed - a seed for the random selection, frcounter - a fast restart counter, the LNS uses and tree search method.

In [1] the authors define two neighborhood selection methods relying on information coming from constraint propagation. The first method defines the set of variable to freeze by incrementally building
a partial assignment , starting from an empty scope. The underlying idea is that of freezing related variables. the volume of domain reduced, it helps to link variables together inside or outside partial solutions.

The algorithm uses the current size of the domains of the decision variables to control the size of the neighborhood, then when a variable is frozen, propagation occurs, by tracing the volume of the domain reduction, we can detect which variables are linked to the frozen variable, this information will be utilized to choose the next variable to freeze, so the algorithm becomes:

Table 2: PLNS guided search method (algorithm source [1])

```
Propagation Guided LNS method.
While fragment size is greater than desired size
        If variable list is empty then
                Choose unbound variable randomly
        Else
                Choose variable in variable list
        End if
        Freeze variable and propagate
        Update variable list
End while
```

RLNS

The LNSFactory provides pre-defined configurations. Here is the way to declare LNS to solve a problem:

```
LNSFactory.rlns(solver, ivars, 30, 20140909L, new FailCounter(solver, 100));
solver.findOptimalSolution(ResolutionPolicy.MINIMIZE, objective);
```

It declares a random LNS which, on a solution, computes a partial solution based on ivars. If no solution are found within 100 fails (FailCounter(solver, 100)), a restart is forced. Then, every 30 calls to this neighborhood, the number of fixed variables is randomly picked. 20140909L is the seed for the java.util.Random. it uses a Random selection scheme of the variables thus it is the slowest method of the previous ones.

While the PLNS gives an outstanding performance with optimal or near optimal results.a

Implementation and Algorithm

Implemented Problem Formulation

The implemented problem has been simplified to include the following variables and constraints

n : number of products to allocate
m : number of shelves
ai : length of facing of product i
Tj : length of shelf j
Li : minimum number of facings required for product i
Ui : maximum number of facings required for product i
xij : number of facings of product i on part k of shelf j
yij = 1 if facings of product i are assigned to shelf j, 0 otherwise
Pi : profit per unit of product i
Cj : priority coefficient of shelf j
Pij = PiCj : profitability of product i if placed on shelf j
i = 1.. n, j = 1 .. m

Objective Function: Maximization of total profit given by :

$$Z = \sum_{i=1}^{n} \sum_{j=1}^{m} Pij.xij \qquad (1)$$

(1) Which means that the summations of all profits per unit product multiplied by the number of facings for a product for all products in the system allocated on all shelves in the system, the maximization of this value is the maximization of the total profit.

Subject to the following constraints:

$$\sum_{i=1}^{n} ai.xij \leq Tj \quad (for\ all)\ j = 1..m \qquad (2)$$

(2) Which says that the summation of the length of all facings of product *i* allocated on a shelf j is less than the length of the shelf for all shelves in the system.

$$\sum_{j=1}^{m} yij = 1 \qquad (for\ all)i = 1..n \qquad (3)$$

(3) This constraint states that a product can lie only on one shelf, thus this Boolean variable is either 1 or 0, it will be 1 if facing I lies on shelf j otherwise it will be 0, thus summation of all these variables will be 1 will make the product shelf positioning condition hold, so for this variable there will be nXm different variables, the summation of all variables in the same column is 1.

$$Li \leq \sum_{j=1}^{m} xij \leq Ui \qquad (for\ all)i = 1..n \qquad (4)$$

(4) The lower bound and upper bound of the number of facings of a given product, is defined by input, all product facing variables are limited by minimum number of facings condition and maximum number of facings condition defined by the input data.

$$\sum_{i=1}^{n} Li.ai \leq \sum_{j=1}^{m} Tj \qquad (5)$$

(5) is a bound for the entire system which means that the input sample actually meets the minimum space requirements, that means, that before going into attempting to solve the problem we need to verify that there is enough space in the shelves to hold the summation of the minimum bounds of the number of facings multiplied by the length of facings in a shelf for all products and all shelves.

For the purpose of implementation and considering that a product is only located on one shelf, the objective can be rewritten as the following

$$Z = \sum_{i=1}^{n} \sum_{j=1}^{m} yij.Pij.xi \qquad (6)$$

where yij will denote the location of product xi.

The Y here will become a 2D array of 0s and 1s, which will become variable array along with xi, thus the problem in (6) is not completely linear, how ever in our implementation, the Choco Solver, can still handle this type of product of the two variables.

The LNS solver will iterate the Binary array as well as scanning the range of the X array between the lower bound and the upper bound to improve the final solution, depending on which approach is applied (RLNS, ELNS, PLNS).

Other solvers cant work the product of two variables (which makes the problem not truly linear), however, in our case, the none linearity is considered as simple as: Product of a Binary Variable with a Bounded variable (and generalizing it for all the variables at hand), delinearization of the problem using constraint programming methods can be done as the following[4]:

Considering: $z = A.x$ $\quad where\ x\ is\ binary, A\ is\ a\ bounded\ variable$

$$min\{0, \underline{A}\} \leq z \leq \bar{A}$$

$$\underline{A}x \leq z \leq \bar{A}x$$

$$A - (1-x)\bar{A} \leq z \leq A - (1-x)\underline{A}$$

$$z \leq A + (1-x)\bar{A}$$

The delinearization can be described using 7 constraints as the following.

$$z \leq A_{Upper_{Bound}}$$

$$-z \leq 0$$

$$z \leq A_{Upper_{Bound}}.x$$

$$-z + A_{Lower_{Bound}}.x \leq 0$$

$$z - A - A_{Lower_{Bound}}.x \leq A_{Lower_{Bound}}$$

$$-z + A + A_{Upper_{Bound}}.x \leq A_{Upper_{Bound}}$$

$$z - A + A_{Upper_{Bound}}.x \leq A_{Upper_{Bound}}$$

Delinearization is only needed for solvers that cant handle the product of two variables.

In the implementation, extra variables have been created for the problem to allow the previous formulation to be adapted to the Choco solver and to create a successful solution.

Solver Choice

In our implementation of the above problem we have chosen the Choco Solver, which is Free Open Source Java library dedicated to Constraint Programming by modeling complex problems in a declarative way mainly depending of the the set of constraints that need to be satisfied in every solution. Then, the optimization problem is solved by alternating constraint filtering algorithms with a search mechanism tell a certain criteria is satisfied or an optimal solution is found.

Implementation Classes and Functionality Distribution

Implementation is mainly distributed into three main sections:

1. Modeling the problem, solving, and solution iteration
2. Planogram drawing and motion
3. GUI interaction and information processing using multithreading operations.

The above three functionalities are distributed amongst the following classes and according the following class diagram (the diagram shows the most important functions, not all, as they are hundreds)

Table 3: Classes and description

Class Name	Class description
Data	Contains all information about the products and shelves extracted after parsing a data file

ParseData	Contains methods for reading a data file containing our product/shelf information from a file stored in the file system or from a resource, the data read into a String.
	The class also contains a method for parsing the resulting data string into a Data object.
	For the purpose of our application we have created a data file set, the file format is in the following form. (sample)
	9 9 160
	product1 family1 1 5 5 100 2.55
	product2 family1 2 10 5 100 3.22
	product3 family1 3 6 5 100 2.55
	product4 family2 4 15 2 100 3.55
	product5 family2 5 20 2 100 3.45
	product6 family3 6 5 5 100 2.1
	product7 family3 7 40 2 100 4.03
	product8 family4 8 15 5 100 2.55
	product9 family5 9 8 5 100 2.55
	1 1 1.88 2
	2 1 1.88 1
	3 1 1.65 2
	4 2 1.65 2
	5 2 1.45 1
	6 2 1.45 2
	7 3 1.3 2
	8 3 1.3 1
	9 3 1.25 2
	The first line of the file has the format: #N M S
	#N: first number indicates number of products, thus parser should read N lines
	#M: second number indicates number fof shelves, thus parser must read M lines afterwards
	#S: third number (width of a shelf in cm)
	#first N lines have the following format
	#prodcut_name family_name ID(int) facing_length_in_cm(int)
	minmum_facing_required(int) maximum_facing_required(int) profit(double)
	#second M lines have the following format
	#ID(int) shelf_id(int) priority_co_shelf(double) priority_co_part(double)

Chart	Chart Panel class that allows the application to draw a data set of two points (solution objective value vs solution number) to show the improvement of the solution.

SSAP	The core class for solving the SSAP problem, passing it the data object, it attempts to find and improve the solution of the given problem, more details about this class will be shown in a separate section
StopCriteria	A class to contains a stop criteria, which could be utilized by the GUI for instance to stop the search.
SSAP_MainUI	The main GUI component that presents all data and options to the user, it shows tables of the data file loaded (products/shelves data) and also shows solution related numerical data as the solution progresses, including current shelf occupancy and the profit factor gained by current solution. The GUI also initiates the chart drawing and the planogram drawing and allows saving the planogram into a JPG file at any given point in execution. The entire application runs in multithreaded way, as the solver runs in background thread, same goes for the chart and planogram drawing, not much attention is paid for the synchronization amongst the threads though, however the usage of threads allows a smooth graphical and information presentation experience.
PlanogramUI	A frame that allows drawing of planograms, which contains two scrollable panels that allows the full view of the entire data without anything lost from the view. First panel draws the products and their colors. While the other panel draws the shelves and products allocated on them, showing the shelf ID and product ID and color, which allows the user to visually see how the allocation happens and apply it directly in his/her store. The drawing of planograms uses double buffered images that shows the solution changing and products allocations in smooth video like view. This class also allows saving the planograms in jpg file format
ProductDrawingData	Contains basic information about product drawing unit, including a drawing function that draws all facings of the same product onto a graphics object.
PlanoGraphics	Contains the buffered images of the planograms, and all related drawing information, based on simple data such as shelf width and count and count of products, it creates buffered images and then it draws the entire shelves and products which are created at an earlier stage and converted into a ProductDrawingData list. Shelves are drawn first using shelf drawing methods, the products are drawn on top the of the shelves using product drawing methods. Drawing process is organized in such a way that each object knows how to draw itself over the graphics object in unison to present the final drawing.
ShelfDrawing	Contains the shelf drawing information including width height, id, color, and location… A shelf always knows how to draw itself over a graphics object.
ProductDrawing	Contains all product related drawing information including id, color, location…. A product drawing always knows to draw itself over a graphics object.

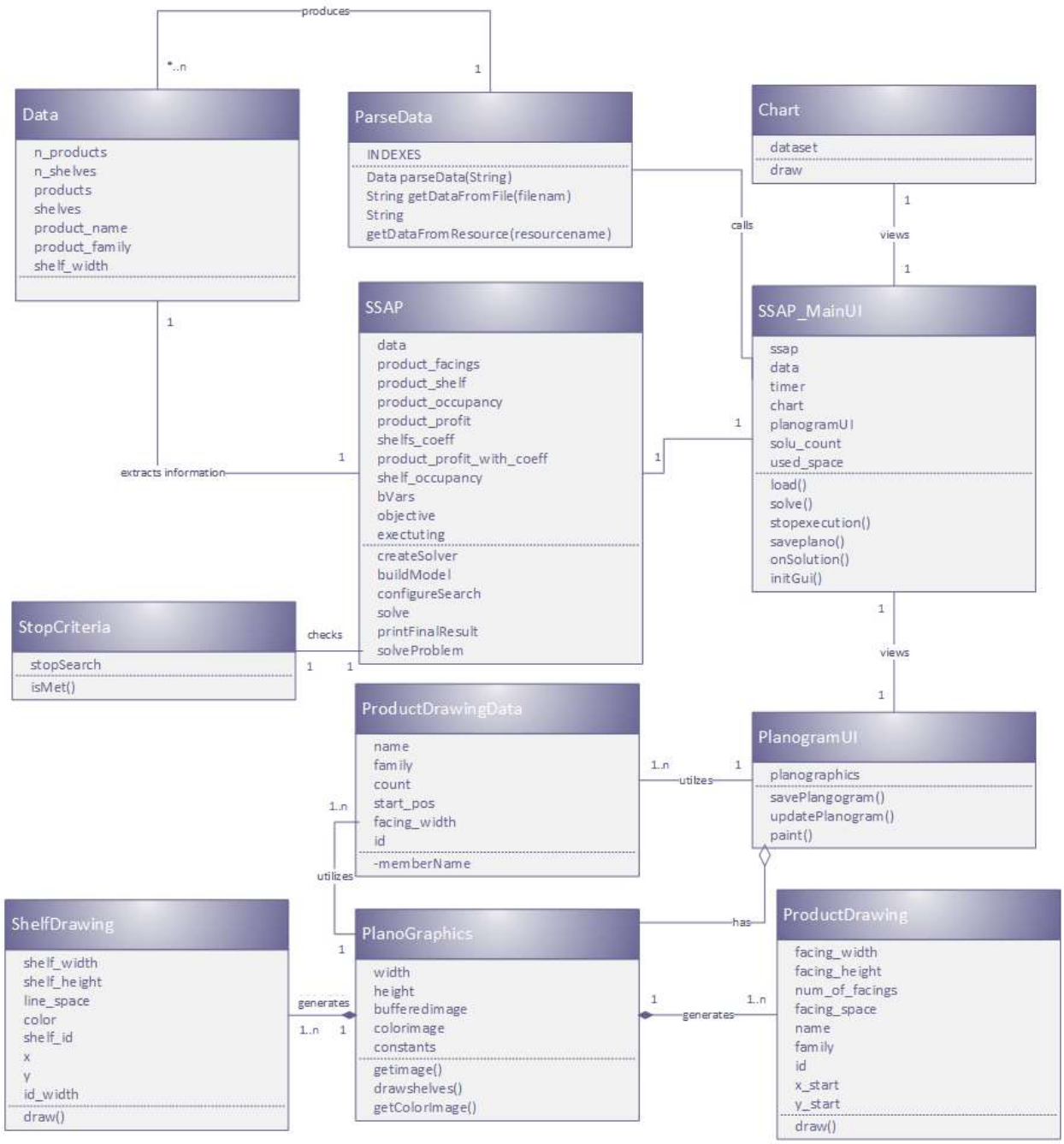


Figure 2: partial class diagram view

Planogram drawing concepts and method

The planogram is drawn as mentioned before using double buffered images, and consists of several components and several layers for drawing where each component knows how to draw itself within the given boundaries and information, the sequence of drawing the planograms happens as the following:

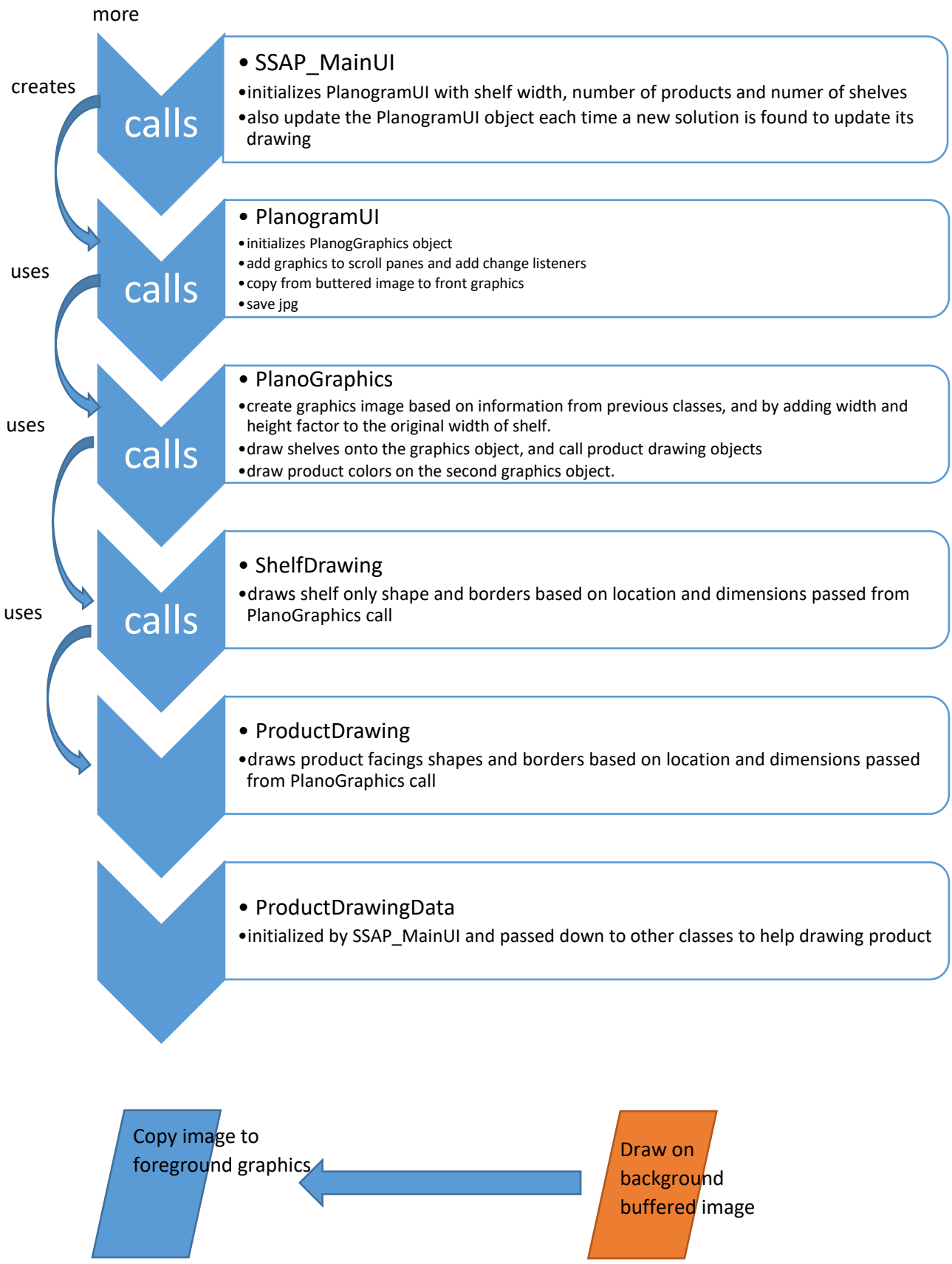


Figure 3: drawing planograms steps and method

As the planogram may contain hundreds of visual elements all to be drawn separately, the wise choice of drawing for smooth visualization is the use of double buffering images, where we draw on the background image then copy it one time to the front screen image when change occurs.

Solver working steps.

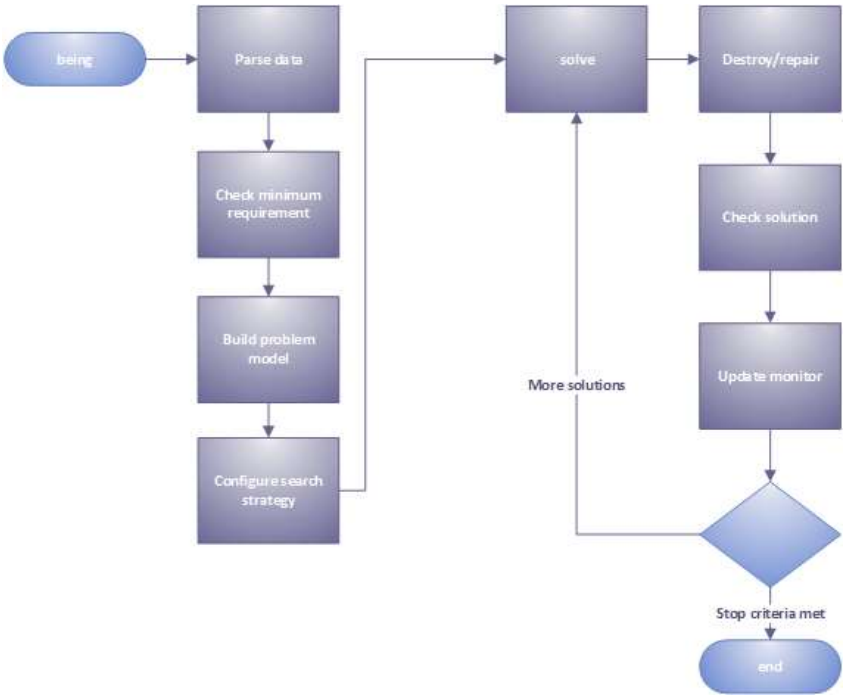


Figure 4: solver working steps

In this section we will expand the explanation given earlier about the SSAP class and the steps of modeling and solving the problem.

Solver mainly works by defining variables and constraints over these variables, these variables are located within connected equations same as mentioned in a previous section about the mathematical model.

The solver running process consists of the following:

- Calculate minimum space allocation problem: which make sure that the problem can be solved to begin with, that is there is enough space in the shelves that can allocate the minimum number of faces of all products.
- Creating the solver
- Building the model:
    - Create and initialize variables which includes:
        - Number of product facings variables (n), which is a bounded variable with the minimum and maximum number of facings obtained from the data set.

        - Product-shelf pair variables (n): which is bounded by 0 and the maximum number of shelves in data set.
        - Product-occupancy variables (n): which is a scaled version of the product facings variables, scaled by the facing length value (a scaled view)
        - Product-profit variables (n): which is a scaled view of the product facings variable with the scaling factor as being the profit value extracted from the data set (scaled view)
        - Product-profit-shelf-coeff variables (n) will be used for constraints purposes which is a none linear variable resulting from the product of product_profit and the shelf coefficient and finally used in the objective function, where the objective becomes the summation of the variables of this type.
        - Shelf-occupancy variables (n): how much a shelf is occupied, bounded by 0 and the length of the shelf.
        - Shelf_coffecient (n) : the coefficient of the shelf a product is lying onto.
        - Shelves Boolean variable (n X m) (which indicate whether a product I is located on shelf j), will also be used with constrains to impose the product_shelf rules, and that a product should only be located on one shelf.
        - Boolean variables (will contain a list of Boolean variables in the system), fattening the previous Boolean variables into a list.
        - Objective variable (total profit): which is the summation of the none linear variable created earlier of the product profit and the shelf coefficient.
    - Constraints:
        - Constraints are modeled based on the formulas mentioned earlier, over the variables earlier.
            - shelf_prod_coeff_cons constraint: makes sure that shelfs_coeff[i] =coeffs[product_shelf[i]), where shelfs_coeff[i] and product_shekf[i] are variables, while coeffs is an int array.
            - prod_with_coeff_profit_cons constraint insures that product_profit_with_coeff[i] variable is the product of product_profit[i] and shelfs_coeff[i] variables.
            - c1 constraint insures that product_shelf[i] = j if shelves[i][j] is 1
            - c2 constraint insures that product_shelf[i] != j if shelves[i][j] is 0
            - previous two constraints are connected via if then else constraint

- prod_on_one_shelf_cons constraint insures that summation of shelves[i] variables is 1, meaning that product lies on one shelf only.
- prod_shelf_occupany_cons constraint insures that prod_shelf_occupany[i] is the result of the product of product_occupancy[i] and shelves[i][j] variables
- shelf_cons insures that the summation of prod_shelf_occupany variables array equals to shelf_occupancy[j], thus the occupancy of single products on the same shelf are equal to the shelf occupancy.
- The final constraint is that the summation of product_profit_with_coeff variables equals to the objective variable.

        - Constraints are carefully chosen and the slightest error or logical error will result in wrong results or in failure of finding a result.

- Configuring search strategy
    - Defines the search strategy, where we define the main strategy as the randomization of Boolean variables which indicate that the shelf-product ij pair presence, this value will be randomized to find solutions.
    - Lexico_UB attempts to randomize the occupancy numbers from larger numbers of the upper bound and then changing them to improve the solution which usually yields a good solution from the first attempt and faster search results.
    - Lexico_LB attempts to use the lower bounds and increase the occupancy numbers up, which usually yields a slow search
    - We also add the stop criteria here (which is manual stop in our implementation if optimal is not found), we also define time limits for the solver to stop after some time of running equals to 3 times the number of products in seconds.
- Listen to solver changes
    - To provide log data to see what is happening.
- Solve.
    - The objective function is to Maximize the profit.
    - Uses one of three methods:
        - PLNS : Propagation based (best method and fastest)
        - RLNS: Random based
        - ELNS: Explanation based.

Test Cases and Results.

In order to test the solver, input data files have been created to simulate a possible scenarios of shelves and products.

(the application will have the option to load a customized external file other than those added to the resources and are part of the application, it is important to use similar files as those attached

in the same formatting and spacing and line intend, as strengthening the data file parser or data input is not our main concern in this project, we only need to present a way to test the different data files over our implementation)

A sample data looks like the following for products (Family is not utilized in out implementation, it could be fatherly investigated in the future to create more aggregation rules)

Table 4: sample product data table

product Name	Family	Product ID	Legth of facing	minimum facings required	maximum facings required	profit
product1	family1	1	5	5	100	2.55
product2	family1	2	10	5	100	3.22
product3	family1	3	6	5	100	2.55
product4	family2	4	15	2	100	3.55
product5	family2	5	20	2	100	3.45
product6	family3	6	5	5	100	2.1
product7	family3	7	40	2	100	4.03
product8	family4	8	15	5	100	2.55
product9	family5	9	8	5	100	2.55

As for shelves, it looks like this (however we only utilize the ID, and Shelf Coefficient), the remaining fields could be extended or utilized in more complicated scenarios.

Table 5: sample shelf data table

ID	shelf ID	priority coefficient of shelf	priority coefficient of part
1	1	1.88	2
2	1	1.88	1
3	1	1.65	2
4	2	1.65	2
5	2	1.45	1

The above data is combined into one text file of the following format (first line contains three numbers, indicating, number of product, number of shelves, length of shelf, respectively)

Then we read numbers of lines matching the first number, then number of lines matching second number, all data are separated by tabs or spaces (including first string of each line which starts with a tab or space), the parsing is straight forward for this file using Scanners.

Table 6: sample data file

```
9 9 160
        product1        family1         1       5       5       100     2.55
        product2        family1         2       10      5       100     3.22
        product3        family1         3       6       5       100     2.55
```

	product4		family2		4	15	2	100	3.55
	product5		family2		5	20	2	100	3.45
	product6		family3		6	5	5	100	2.1
	product7		family3		7	40	2	100	4.03
	product8		family4		8	15	5	100	2.55
	product9		family5		9	8	5	100	2.55
	1	1	1.88	2					
	2	1	1.88	1					
	3	1	1.65	2					
	4	2	1.65	2					
	5	2	1.45	1					
	6	2	1.45	2					
	7	3	1.3	2					
	8	3	1.3	1					
	9	3	1.25	2					

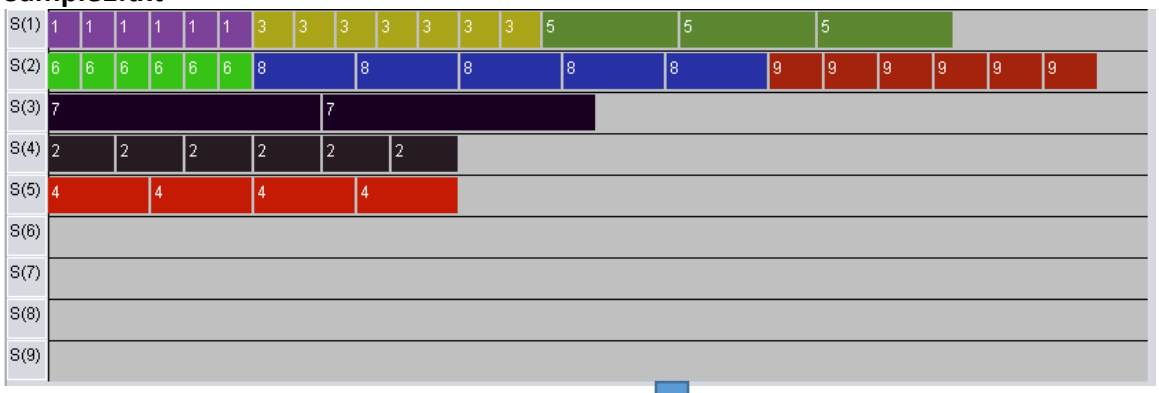
Test Results.

We attach 7 sample data files with the following summary, with the same data format as explained previously.

Table 7: data files in the test cases

File name	Number of products	Number of shelves	Shelf length
Sample1.txt	9	9	160
Sample2.txt	9	5	160
Sample3.txt	20	9	160
Sample4.txt	35	14	160
Sample5.txt	60	20	160
Sample6.txt	130	50	160
Sample7.txt	250	90	160

Sample1.txt



Above picture for one of the first solutions
Bellow is for a for a more advanced solution

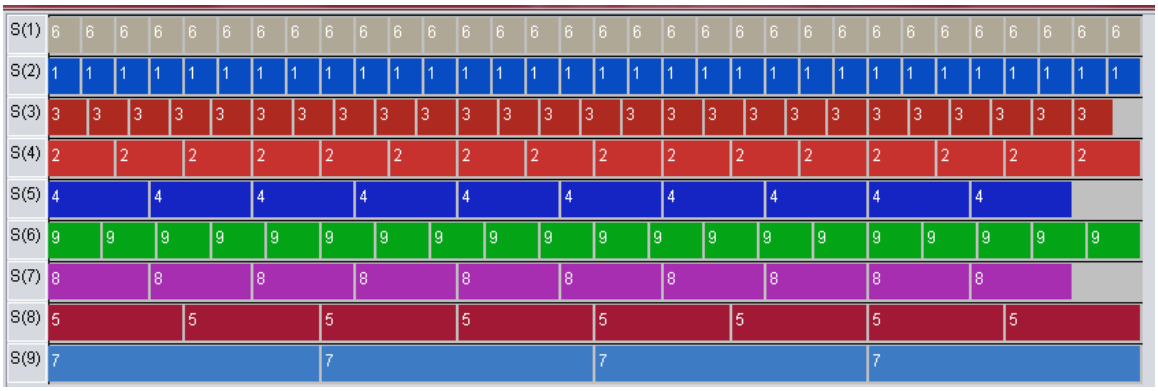


Figure 5: Planograms of a sample problem

the picture above shows how the solver iterate the solution spaced from an initial solution of the problem to the optimal (or near optimal) solution of the problem.
The solver finds that the best solution is to located each product of the 9 on a separate shelf (of the 9 shelves), and that the products with the smallest face width, and good profit margin should be located on the best shelf (which are shelf 1 and 2 according to the priority coefficient values), while the products with large space width (even if they have a good profit, but in comparison to the width, the overall profit becomes less than small products), thus the product 7 was located on the last shelf (which has the worse coefficient value)

The UI allows making a choice of different approaches for the solver to follow (RLNS-PLNS-ELNS), also the strategy at which where the solver begins iterating the variable values from using Lexico_UB and lexico_LB
Lexico_LB: the solver begins iterating the product number of facings from the lower bound and increasing it to improve the objective (which sometimes might take a long time) depending on several criterias (such as occupancy)
Lexico_UB: begins by giving high values for product number of facings which starts with a solution that fills the entire shelves, and begins changing the facings and locations to improve it, thus Lexico_UB, usually reaches better solutions faster in our examples.

Implementation screens with different viewpoints of the project (data, solution improvement diagram, the planograms for the products and shelves, as well console debugging data showing number of total variables and other data…), as the following figures show:

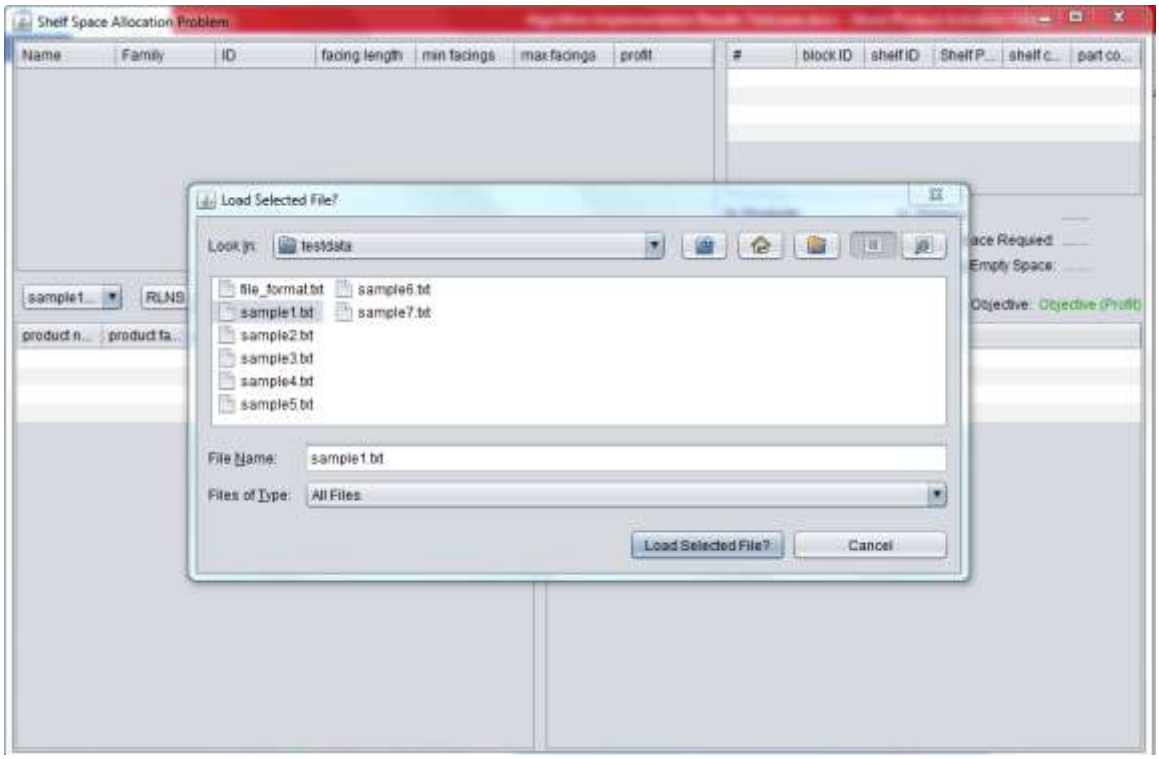


Figure 6: either open a file or choose one from the ready samples

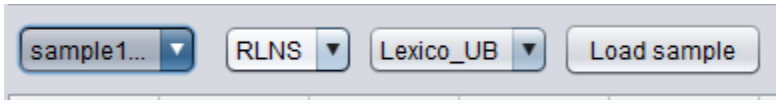


Figure 7: or choose a sample and press Load Sample button before running

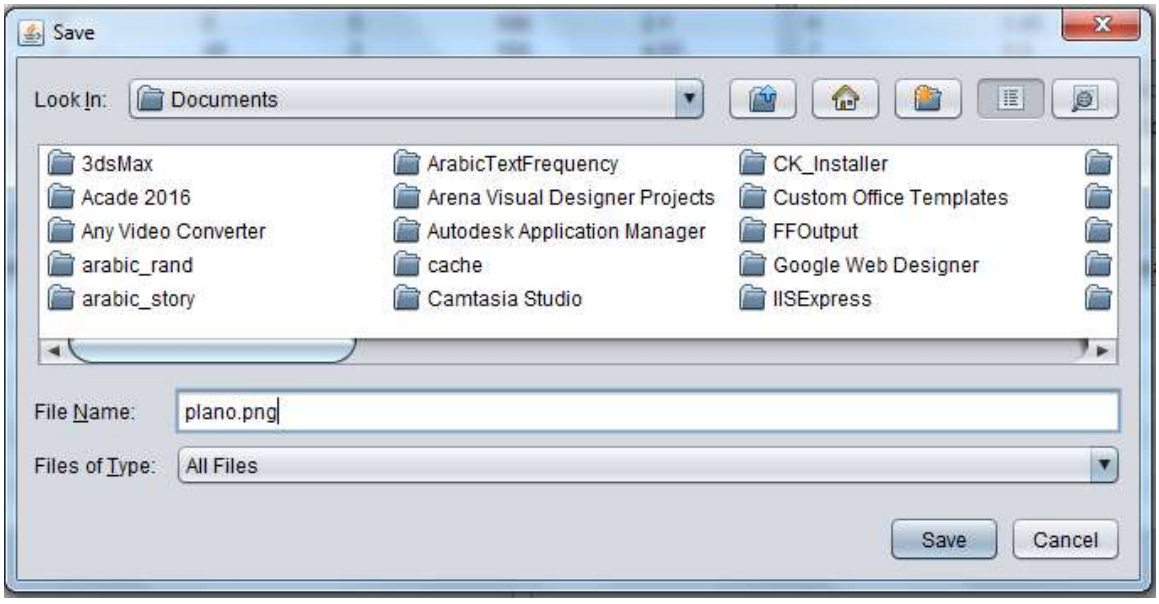


Figure 8: saving the planogram to a file plano.png

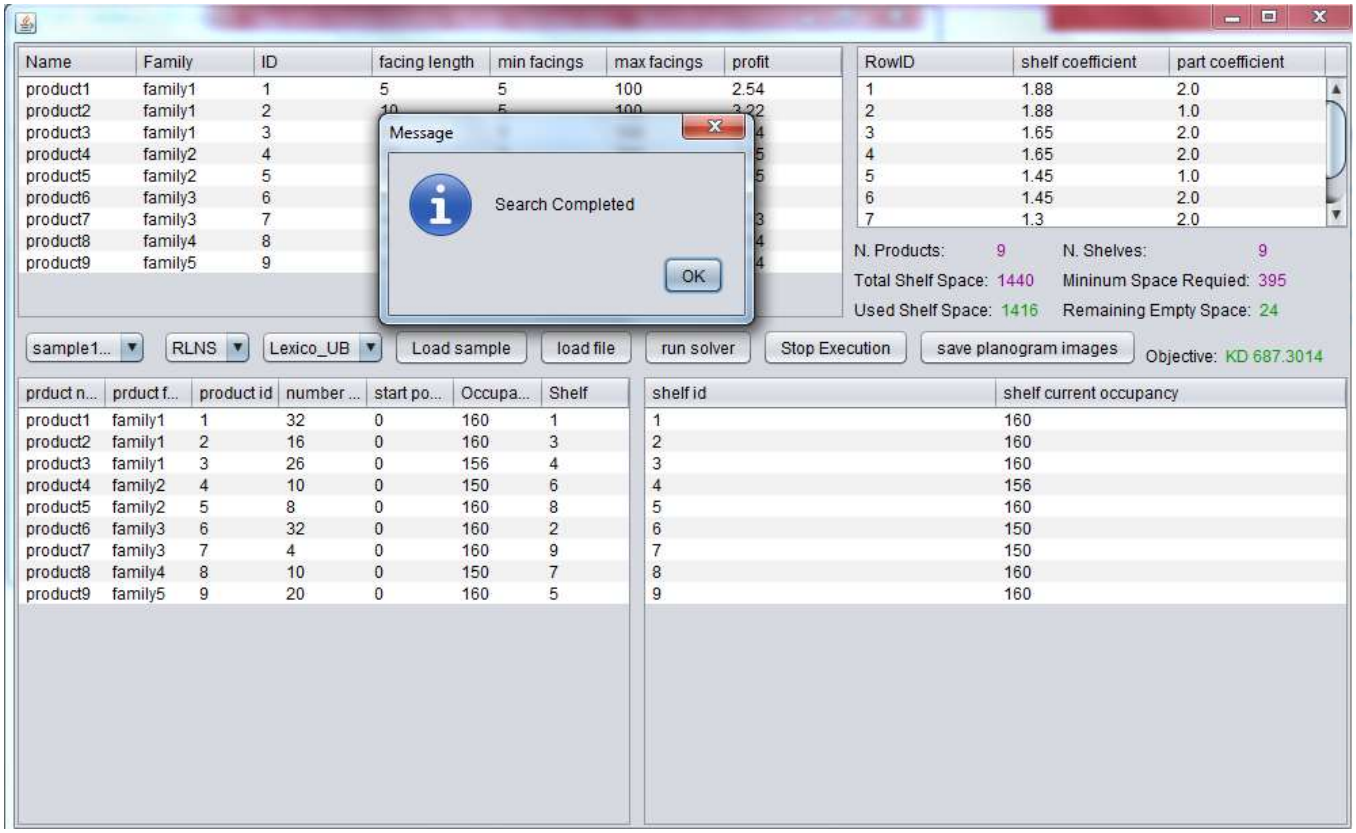


Figure 9: Main UI of the problem, showing input data parsing, output solutions, updated shelf occupancy and product allocation, as well as objective value in Kuwait Dinar, and finishing search message after 27seconds.

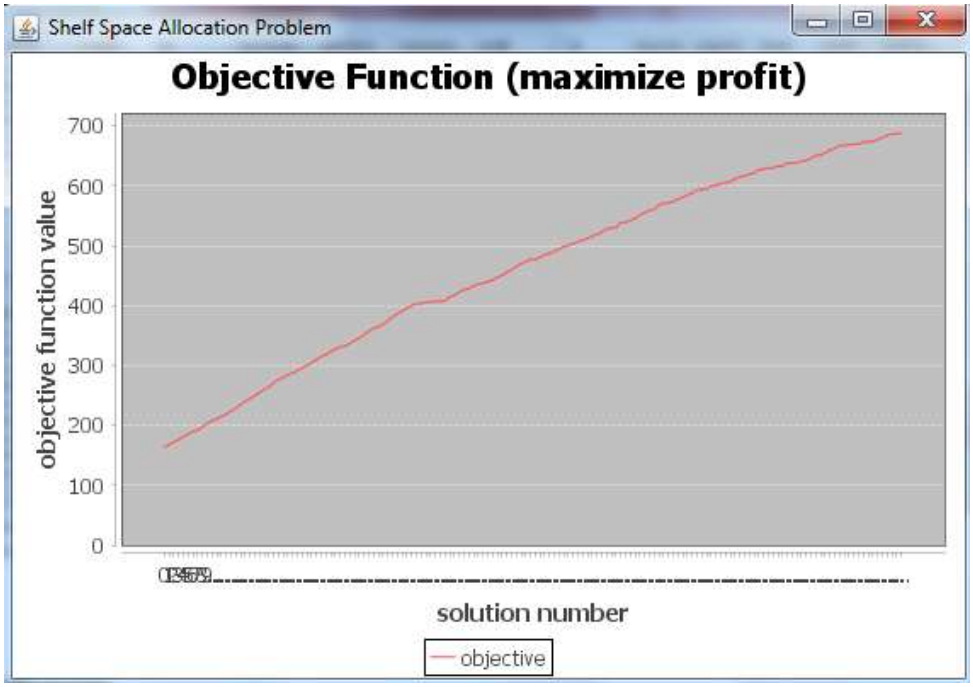


Figure 10: Sample1.txt, Objective function, improving profit reaching optimality

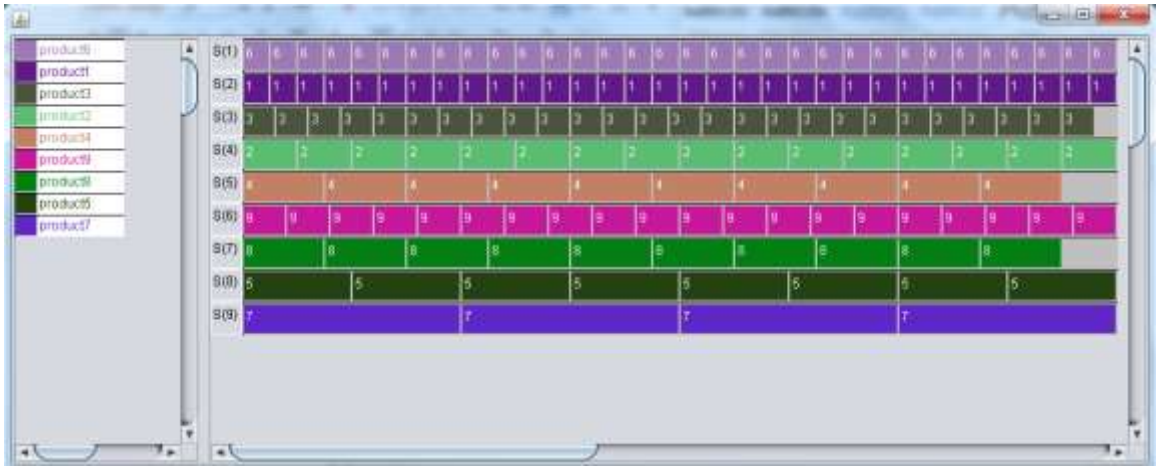


Figure 11: Planogram, showing shelves, products labeled, each product colors are randomized, panel is scrollable, and double buffered for performance

Figure 12: Sample2.txt, no space remaings in the shelves at all(only 1cm).

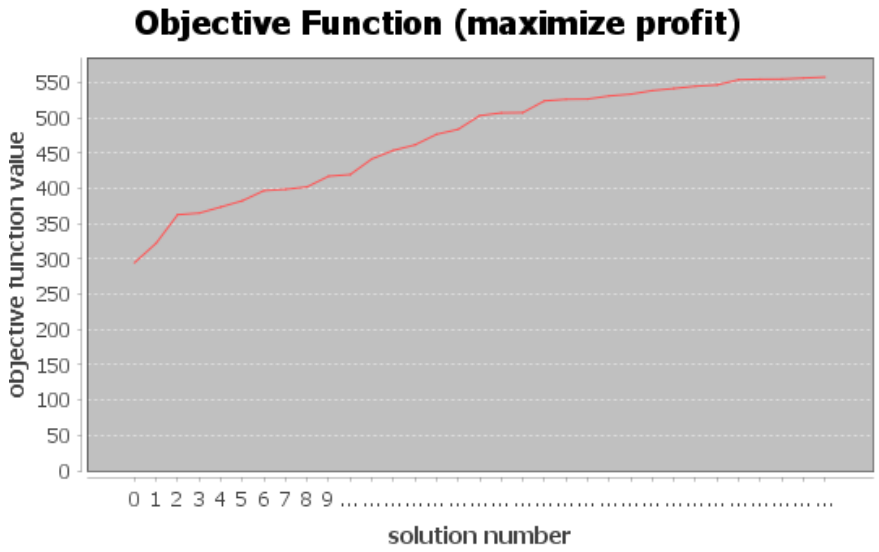


Figure 13:Sample2.txt objective function.

Notice that for Sample2.txt we get the following final data

```
Solver[Shelf Space Allocation Problem]
Solutions: 33
Maximize objective = 5575324,
Building time : 0.388s
Resolution time : 154.443s
Nodes: 1,470,933 (9,524.1 n/s)
Backtracks: 2,914,385
Fails: 1,449,917
Restarts: 15,836
Variables: 377
Constraints: 420
```

Figure 14: sample2.txt, solution data, variable data, total number of constraints and variables, objective should be divided by 10000 to get the right number, as we played around the solver to treat real numbers as integers and because of the limited functionality of the solver.

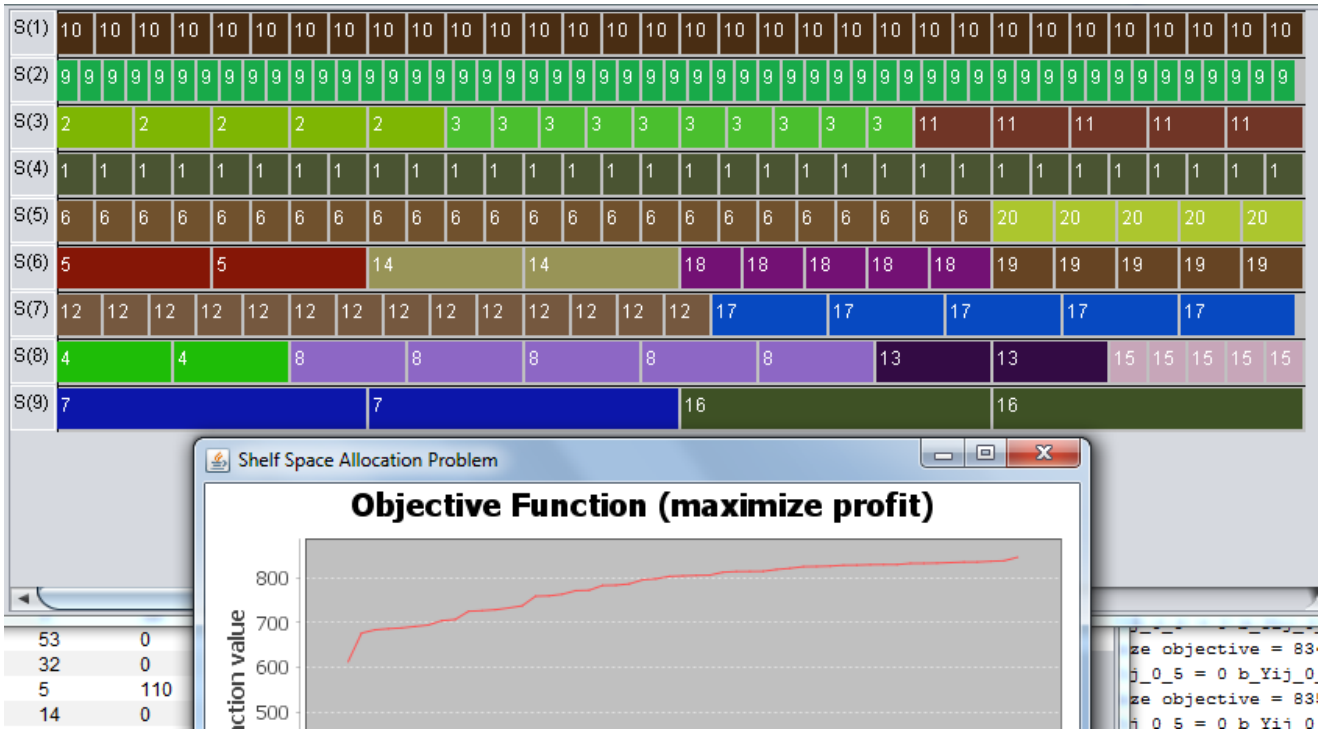


Figure 15: Sampl3.txt, solutions given after 175 seconds, using RLNS, Lexico_UB

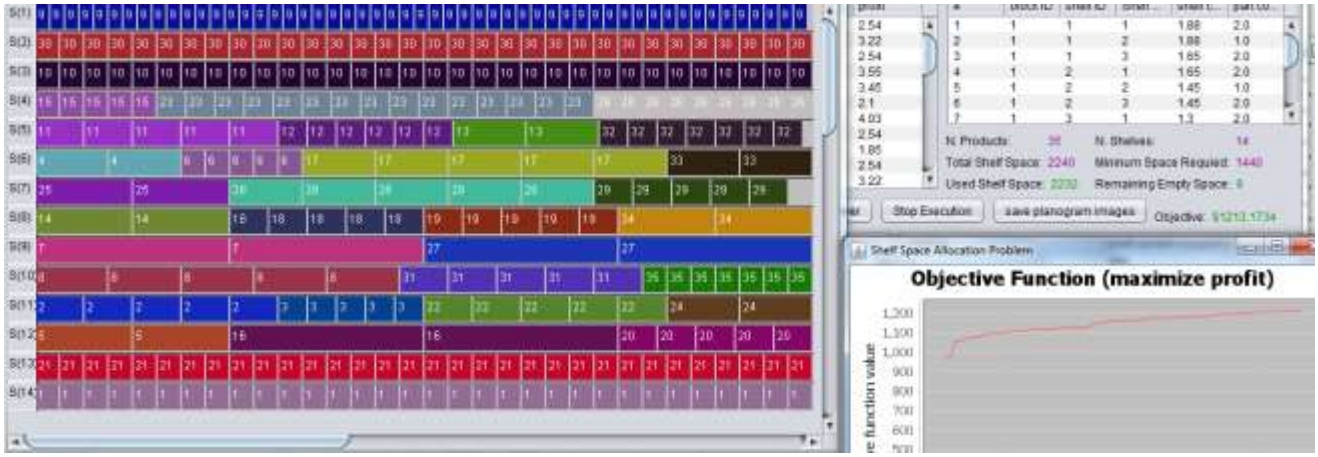


Figure 16: Sample4.txt, using PLNS, Lexico_UB, fter 150seconds, solution can still improve giving the solver more time to find better solutions

The bigger the data file gets, the more time it takes to improve the solution.

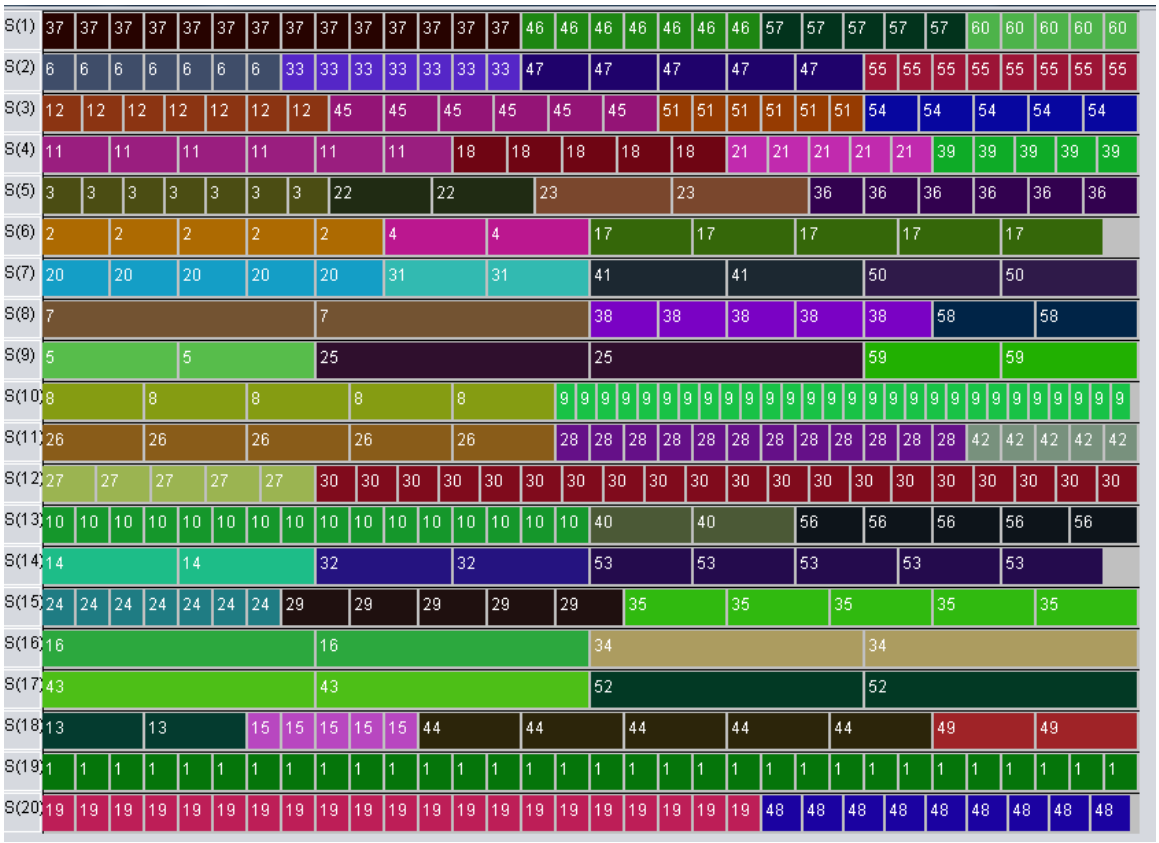


Figure 17: sample5.txt initial planogram after 300 seconds of running the solver, solution can improve greatly in this case but will take longer time

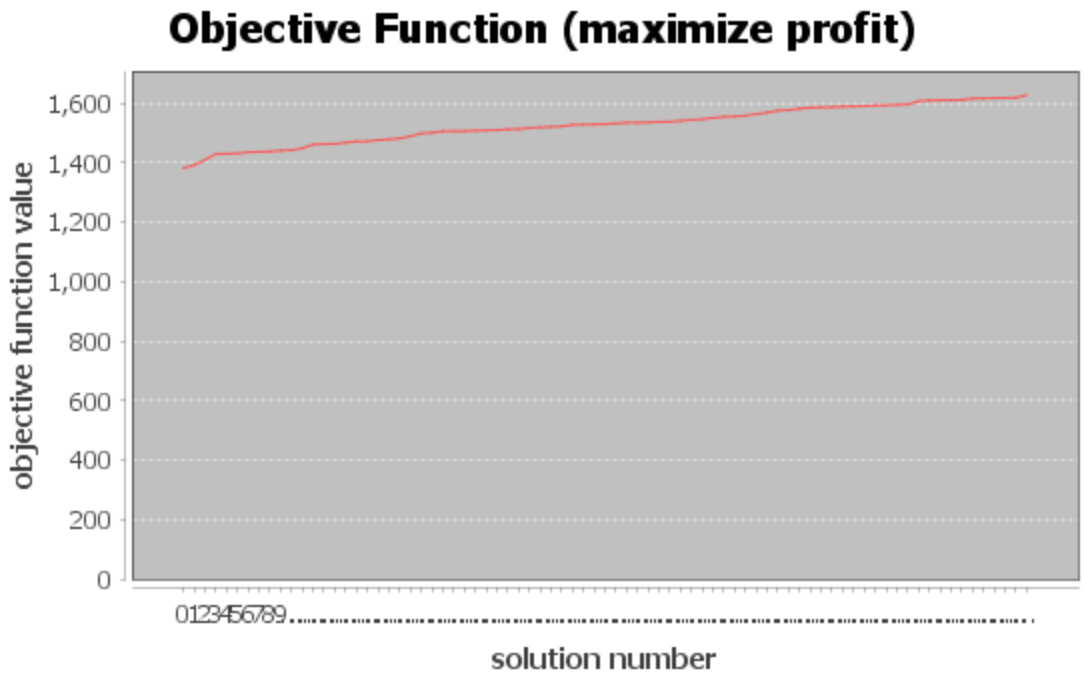


Figure 18: sample5.txt objective function improvement over 300.

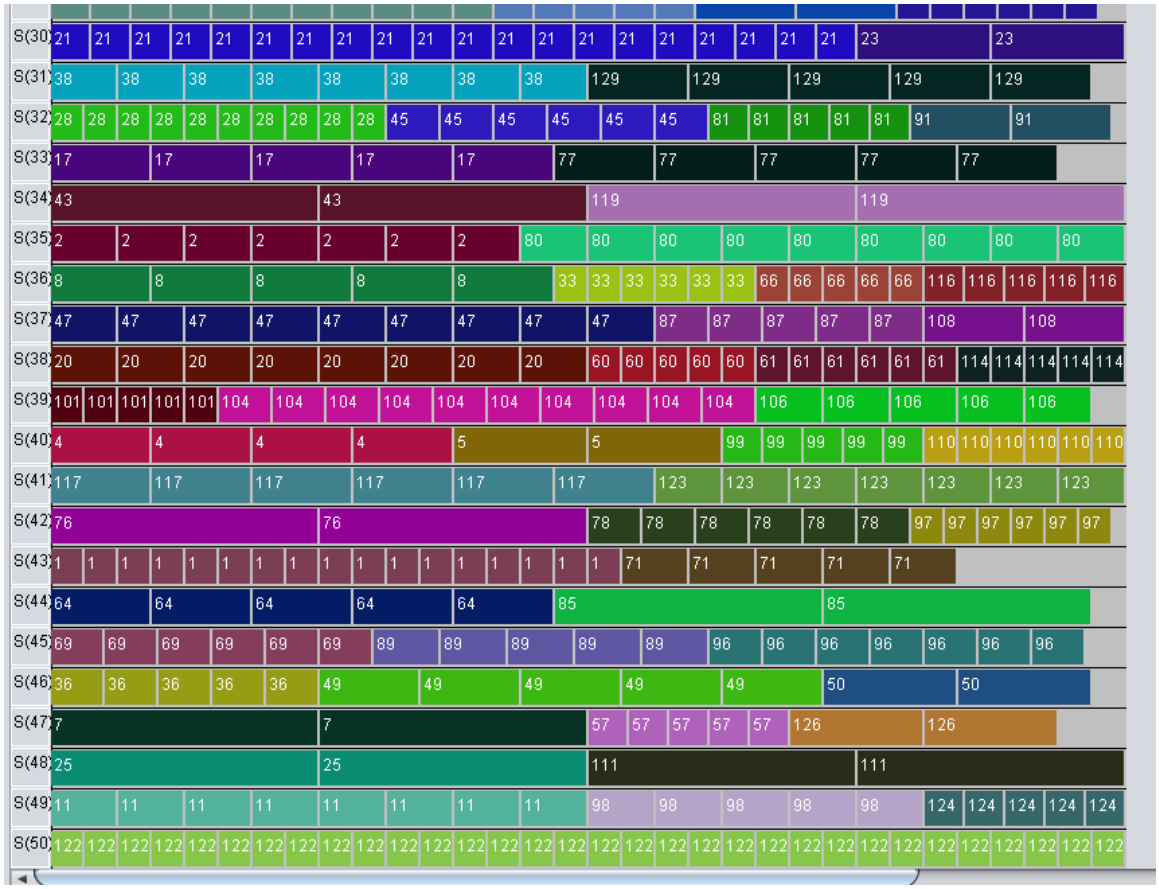


Figure 19: Sample6.txt planogram after 618seconds of running and 38 initial viable solutions

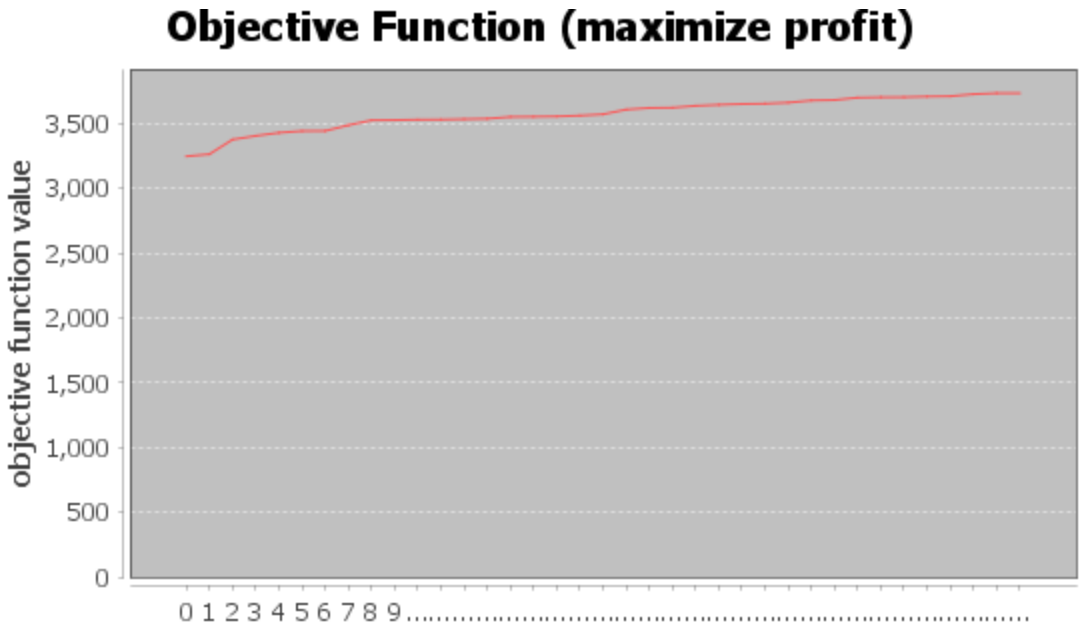


Figure 20: sample6.txt objective improvement over 618seconds.

```
- Solution #38 found. Solver[Shelf Space Allocation Problem], 3
        b_Yij_0_0 = 0 b_Yij_0_1 = 0 b_Yij_0_2 = 0 b_Yij_0_3 = (
- Incomplete search - Limit reached.
        Solver[Shelf Space Allocation Problem]
        Solutions: 38
        Maximize objective = 37380088,
        Building time : 2.621s
        Resolution time : 618.533s
        Nodes: 95,797 (154.9 n/s)
        Backtracks: 116,081
        Fails: 58,000
        Restarts: 580
        Variables: 46,333
        Constraints: 52,831
Shelf Allocations
product 0 [on Shelf:42] Num Of Facings:17
product 1 [on Shelf:34] Num Of Facings:7
product 2 [on Shelf:24] Num Of Facings:10
product 3 [on Shelf:39] Num Of Facings:4
product 4 [on Shelf:39] Num Of Facings:2
```

Figure 21: stopping the solver manually after 618 sconds, the solution can improve greatly if the program continues to look for more solutions.0

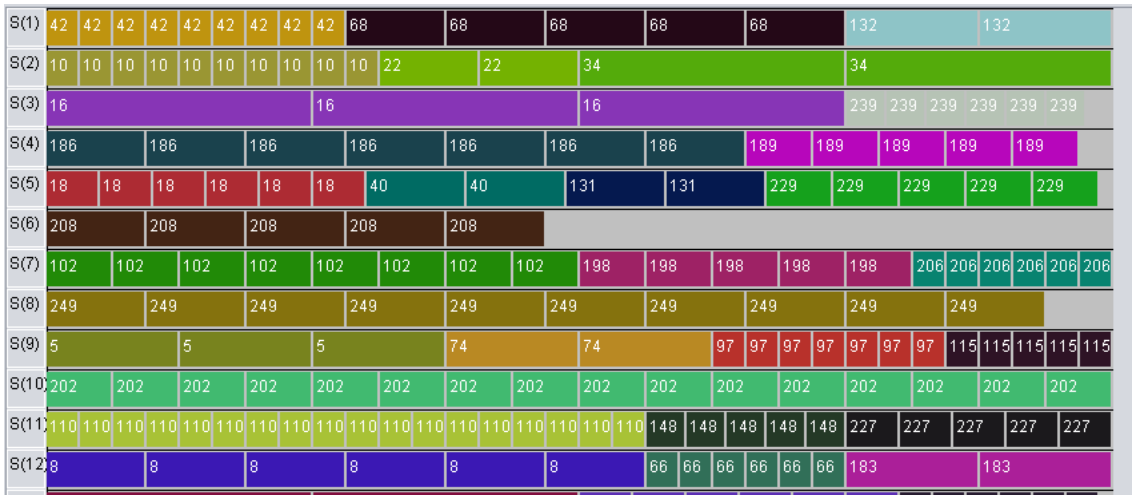


Figure 22: sample7.txt initial planogram after three solutions within 450second time frame.

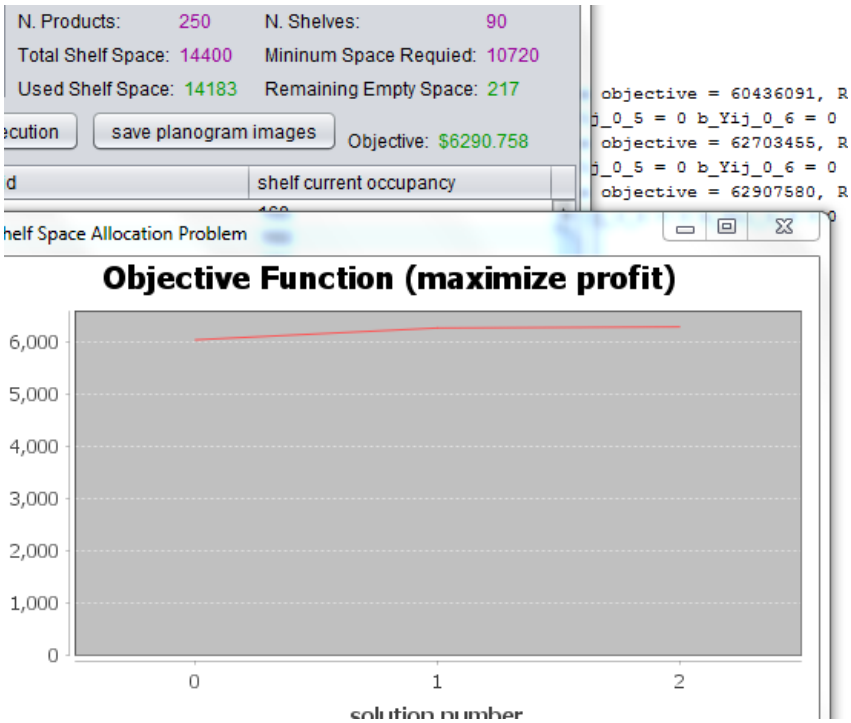


Figure 23: sample7.txt objective function after three solutions within 450seconds of time frame.

Conclusion

We have presented a study for shelf space allocation problem, and suggested a simple model to solve the problem in general, and proved that it can be solved efficiently using Largest Neighborhood Search Based methods, the implementation depended on Choco solver and presented a detailed GUI with a real time planogram frames to watch how the shelves are being sorted out and searched for better or optimal solution within the criteria's given.

References.

[1]. Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In Mark Wallace, editor, Principles and Practice of Constraint Programming - CP 2004, volume 3258 of Lecture Notes in Computer Science, pages 468{481. Springer, 2004. (also found in google books https://books.google.com/books?isbn=3540302018 *)*

[2] Explanation-Based Large Neighborhood Search, Charles Prud'homme, Xavier Lorca Narendra Jussien, 2013

[3] Principles and Practice of Constraint Programming - CP 2004: 10th International Conference, 2004

[4] Coelho, L. (2013, January 7). Linearization of the product of two variables. Retrieved February 25, 2017, from http://www.leandro-coelho.com/linearization-product-variables/

[5] Pisinger, D., & Røpke, S. (2010). Large Neighborhood Search. In M. Gendreau (Ed.), Handbook of Metaheuristics (2 ed., pp. 399-420). Springer. From http://orbit.dtu.dk/files/5293785/Pisinger.pdf