A Benders approach for the constarined minimum break problem

1 Constrained minimum break TTP

1.1 Introduction about Traveling Tournament Problem(TTP):

The Traveling Tournament Problem is a mathematical optimization problem that has been widely studied over the years. The problem can be explained as below.

Given n teams, assuming n to be even, a *round-robin tournament* is a tournament in which every team plays with the other team once. Such a tournament has $n-1$ slots in which $n/2$ games are played. For any game played at a location, one team is called a *home team* and its opponent is the *away team*. The home team is the team from that location itself, where as away team is the one who visited that location to play the match. A *double round-robin tournament* has $2(n-1)$ slots during which each team plays twice with all the other teams in the tournament, once as a *home* team and once as a *away* team.

Distances between team sites are given by an n by n distance matrix D. When a team plays an away match, it is assumed to travel from its home site to the away venue. Consecutive away games for a team constitute a *road trip*; consecutive home games are a *home stand*.

The *Traveling Tournament Problem (TTP)* is to find a double round-robin tournament tournament on the n teams such that, the length of every home stand and road trip is between given upper and lower bounds, and the total distance traveled by the teams is minimized.

In addition to the above mentioned constraints, there may be additional requirements on the solution. One of the variants is to find a *mirrored* double round-robin tournament where in the schedule has a round robin tournament in the first $n-1$ slots are then have the same tournament with venues reversed in the second $n-1$ slots.

1.2 Place constrained Minimum Break:

Each team has a *home-away pattern* (pattern) which is an array of zeros and ones with for each slot(Ex: 101010). A 0 means team plays away and 1 means that team plays home. A set of patterns for all the teams in a tournament is called a *home-away pattern set* (pattern set) and it is said to be feasible if a corresponding *timetable* exists for that particular patter set. A timetable is a matrix with a row for each team and a column for each slot where entrance (i, s) gives the opponent of team i in slot s. The below figure shows a feasible pattern set(figure (a)) for a tournament with 6 teams and a corresponding timetable(figure (b)).

Slot	1	2	3	4	5	6	7	8	9	10
p_1	0	1	0	1	0	1	0	1	0	1
p_2	0	1	$\underline{1}$	0	1	$\underline{1}$	0	$\underline{0}$	1	0
p_3	1	0	1	$\underline{1}$	0	$\underline{0}$	1	0	$\underline{0}$	1
p_4	0	1	0	$\underline{0}$	1	$\underline{1}$	0	1	$\underline{1}$	0
p_5	1	0	1	0	1	0	1	0	1	0
p_6	1	0	$\underline{0}$	1	0	$\underline{0}$	1	$\underline{1}$	0	1

(a)

Slot	1	2	3	4	5	6	7	8	9	10
team 1	6	3	5	2	4	6	3	5	2	4
team 2	5	6	4	1	3	5	6	4	1	3
team 3	4	1	6	5	2	4	1	6	5	2
team 4	3	5	2	6	1	3	5	2	6	1
team 5	2	4	1	3	6	2	4	1	3	6
team 6	1	2	3	4	5	1	2	3	4	5

(b)

The *patterns* which does not alternate(Ex: 100110011) but have two consecutive home games or two consecutive away games are said to have *break* in the last of the two slots. The breaks are underlined in the pattern set of above figure for recognition purpose. A good schedule is said to be the one with less number of breaks and for the same reason patterns with consecutive breaks are considered highly undesirable and are discarded. Notice that a feasible pattern set cannot contain more than two patterns without breaks. Also a *place constraint* requirement saying that a team must play home or away in a given time slot is considered. This explains the constraint minimum break problem.

The problem addressed is to find a double round robin tournament for $2n$ teams such that the games are partitioned in $2(2n-1)$ time slots and each team plays one game in each slot. All place constraints must be satisfied, consecutive breaks are not allowed and the total number of breaks must be minimized. Both the mirrored and the non-mirrored cases will be considered and, in the non-mirrored case, two games with same opponents must be separated by at least k time slots for some given value of k.

The sets of teams and time slots are denoted by T and S respectively and I_i^1 and I_i^0 hold the slots in which team i must play home and away due to place constraints.

$$h_{is} = \begin{cases} 1 & \text{if team } i \text{ plays home in slot } s \\ 0 & \textit{else} \end{cases}$$

$$b_{is} = \begin{cases} 1 & \text{if team } i \text{ has a break in slot } s\text{: } h_{is-1} = h_{is} \\ 0 & \textit{else} \end{cases}$$

$x_{ij} \in S$ gives the slot in which team j visits team i.

The problem can be formally stated as Constraint Programming (CP) problem:

$$min \sum_{i \in T} \sum_{s \in S} b_{is}$$

The below are the constraints:

1. $s.t. sequence(1, 2, 3, all(s \in S)h_{is}, 1, 2n - 1) \qquad i \in T$

This is a global constraint which takes six arguments, three integers: $nBMin, nBMax$ and $width$, and three one dimensional arrays: $vars, values$ and $card$. The constraint requires that each team plays exactly $2n - 1$ home games and has a pattern without consecutive breaks since three consecutive slots must contain at least one and at most 2 games.

2. $(b_{is} = 1) \Leftrightarrow (h_{is-1} = h_{is}) \qquad i \in T, s \in S - \{1\}$

Constraint that defines a break.

3. $b_{i1} = 0 \qquad i \in T$

Constraint that sets all break variables for slot 1 to zero.

4. $\sum_{i \in T} h_{is} = n \qquad s \in S$

Constraint that states that in each time slot, exactly n teams play hom.

5. $h_{is} = 1 \qquad i \in T, s \in I_i^1$
6. $h_{is} = 0 \qquad i \in T, s \in I_0^1$

Constraints 5 and 6 make sure that place requirements are satisfied.

7. $(h_{is} = 0) \vee (h_{js} = 1) \Rightarrow (x_{ij} \neq s) \qquad i, j \in T, i \neq j$

Constraint makes sure that team j does not visit team i in a slot where team i plays away or team j plays home.

8. $alldifferent(all(j \in T\ i)x_{ij}, all(j \in T\ i)x_{ji}) \qquad i \in T$

Constraint takes an array of variables $vars$ with index set I as argument and it requires that all variables in $vars$ must be different. Since $vars$ is the array of all games played by a specific team the constraint makes sure that a team does not play more than a single game in each time slot.

9. $(x_{ij} - xji < -k) \vee (x_{ij} - xji > k) \qquad i, j \in T, i < j$

Constraint separates two games played by the same teams with at least k time slots.

10. $h_{is}, b_{is} \in B \qquad i \in T, s \in S$
11. $x_{ij} \in S \qquad i, j \in T, i \neq j$

Notice, that the constraints 1-3 are all constraints for individual patters, 4 is a constraint for the set of patters and 7-9 consider the assignments of games to the pattern set. This partitioning will be used in the solution method.

2 Pattern Generating Benders Approach (PGBA)

2.1 Legacy approach

In order to solve the different variants of a traveling tournament problem, a three phase approach has proven to be very efficient and has been widely used in the literature, though the order of the following three varies.

Phase 1: Generates *home away patters*(patterns) and *pattern sets*.
Phase 2: Finds *timetables* which are assignments of games to time slots.
Phase 3: Allocates teams to patterns.

2.2 Benders Approach

The hybrid Integer programming and Constraint programming (IP/CP) approach presented in the paper also uses the three phase approach but it obtains speedups by restricting the number of patterns generated in the initial phases and it reduces the number of unfeasible pattern sets found in phase one by using Benders cuts.

Benders approach for the constrained minimum break problem treats: Phase 1 as a master problem, which finds a pattern set with minimal number of breaks and Phase 2,3 as subproblems, that check the feasibility of the pattern set.

If the subproblems are feasible, a schedule has been found; otherwise a *Benders cut* is added to the master problem to cut off infeasible solutions in future iterations. This approach is known as *pattern generating Benders approach*(PGBA).

The PGBA decomposes the problem into the following four components:

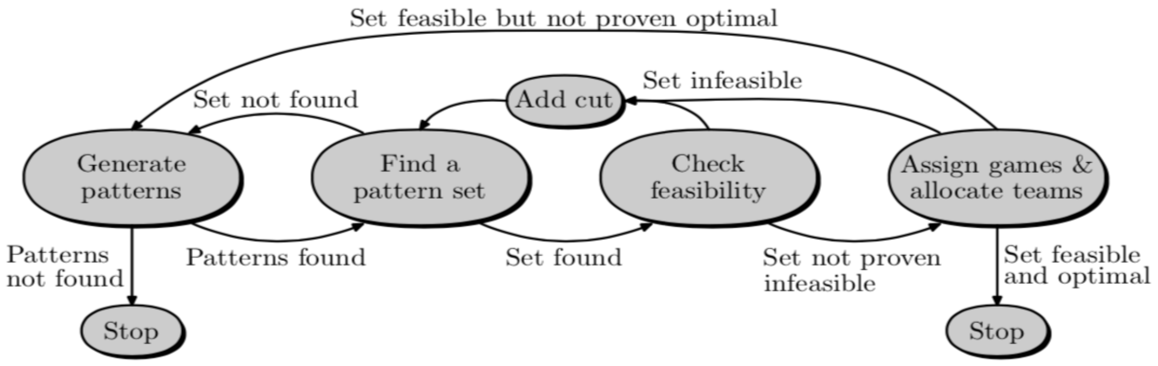
1. Generate feasible patterns
2. Find pattern sets from patterns generated.
3. Check feasibility of the pattern sets and
4. Find a timetable with a feasible team allocation.

However, instead of solving these components one by one, the algorithm iterates between the four components. Contrary to previous approaches, the PGBA only generates patterns with a small number of breaks first. Then, from among these patterns, a minimization problem (the master problem, phase 1) finds a pattern set with a minimal number of breaks.

The, the third component(subproblem 1) of PGBA checks the feasibility of the pattern sets. The component heuristically determines in-feasibility and , when successful, adds Benders cuts to the master problem.

The fourth component(subproblem 2) is used to find a timetable and a team allocation for the pattern sets which have not been proved infeasible. If the pattern set turns out to be infeasible at this stage, again a Benders cut is added to the master problem. Otherwise a feasible solution is found and optimality must be proved.

In case the master problem is infeasible the algorithm tries to generate additional patterns. If extra patterns exist the algorithm continues. Otherwise an optimal solution has been found or in-feasibility of the problem has been proved and the algorithm stops.



The above flowchart gives an overview of the algorithm. Both IP and CP models are used in the algorithm since the decomposition makes it possible to use IP for the optimization problems and CP for feasibility problems.

A CP model is used for the first component of PGBA, generating patters, where the model generates all the patterns starting with an away game and having exactly a given input of say c number of breaks in the output pattern.

The second component(to find pattern sets), outputs the subset of $2n$ patterns called a pattern set such that in each time slot, exactly n teams play home match and the number of breaks in the output pattern set are minimum. Also, in this step a lower bound and upper bound are added to number of breaks to reduce computation and prove optimality respectively.

The third component(feasibility check and Benders cut) uses help of Hall theorem and Hungarian method to get a necessary and sufficient condition for a team allocation to exist. Based on these conditions, a Benders cut is added to second component to cut off the current and similar solutions. Thus a *logic based Benders decomposition* is used instead of traditional Benders decomposition method where the Benders cuts are obtained from a linear subproblem.

The fourth/last component(to find a feasible timetable) uses the output of third component, a pattern set, and checks for the additional feasibility conditions of game assignment. If a feasible game assignment is found the algorithm stops else again a Benders cut is added to the master problem and the above steps are repeated.

The in-detailed explanation for the above mentioned algorithm of different phases is mentioned in section 7 of the paper.

3 Results:

The algorithms are implemented using a script in IBM ILOG OPL Studio and the computational results for different instances of mirrored/non-mirrored have been presented in comparison with a legacy three phase approach. It can be easily recognized that, the PGBA approach mentioned in the paper for solving the place constrained minimum break problem, did extremely well in terms of time taken to solve the problem as n(number of teams in the tournament) increased.

The paper also proposes to solve a known traveling tournament problem called *Constant Distance Traveling Tournament Problem* by modifying the algorithm to maximize the number of breaks instead of minimizing it.

4 My plan:

My plan would be to choose an optimization tool(ILOG OPL CPLEX or GOOGLE OR) and get hands-on experience of the tool from tutorials/sources and implement the algorithms in paper or solve a real life problem/question using the linear and constraint programming mentioned in the paper with which the constrained minimum break problem is solved.

5 References:

1. Scheduling a major college basketball conference, GEORGE L. NEMHAUSER and MICHAEL A. TRICK

2. A Benders approach for constrained minimum break problem, Rasmus V. Rasmussen1 and Michael A. Trick2

6 Implementation:

I had chosen the optimization tool IBM ILOG OPL CPLEX to implement a problem/question(buffet placement) stated in homework 1 of our course using linear/constraint programming.

6.1 Problem statement of Buffet Placement:

Leigh is sorting out the placement of dishes for a conference buffet, and must decide how many tables are needed to place the dishes. Each dish is either hot or cold, and occupies a fixed width of the table. Additionally, a hot and a cold dish may not be placed immediately next to each other; they must be separated by a minimum distance. The venue has only provides one size of table.

Given the number of each dish type to be served, the objective is to find the minimum number of tables necessary to serve all dishes.

Input:

nbDishes = Number of dishes to be placed.
nHot = Number of hot dishes among all the dishes, rest would be cold dishes.
table_width = Width of a table and all the tables have same width.
separation = Separation needed between a cold and hot dish when places next to each other.
dish_size[...] = Array containing the size of each dish.
demand[...] = Array containing the demand of each dish.

Output:

tables_required = Minimum number of tables required to place all the given dishes.

6.2 Linear Programming Formulation:

The above stated problem can be solved by using linear programming. Linear programming is also called Linear Optimization, is a method to achieve the best outcome (such as maximum profit or lowest cost) in a mathematical model whose requirements are represented by linear relationships. Linear programming is a special case of mathematical programming (mathematical optimization).

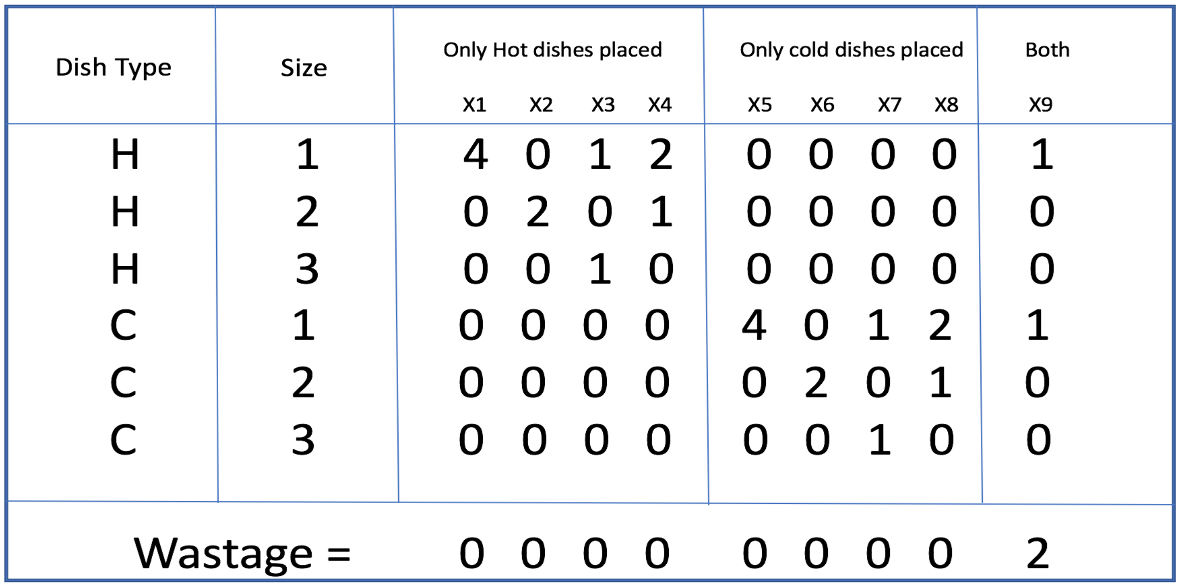
The above stated problem is to place all the cold and hot dishes satisfying given constraint in a way that that the total number of tables required is minimized, nothing but minimizing the wastage of resource. First thing we need to figure out is the wastage (the empty space left on a given table) for any given combination of dishes placed on a table. The other way of saying is to find the different dish placement patterns of wastage of unused space.

Let us assume some input to understand as follows:

int nbDishes = 6;
int nHot = 3;
int table_width=4;
int seperation=2;
int dish_size[1..nbDishes] = [1,2,2,1,2,3];
int demand[1..nbDishes]= [4,2,2,6,1,1];

An example of pattern would be [4,0,0,0,0,0] which means that there can be 4 in quantity of dish number 1, which is Hot and of size 1, and there is no space left for other dishes on this table; wastage is 0. Considering another pattern [1,0,0,1,0,0] which says there can 1 hot dish and 1 cold dish of each sizes 1 on a single table of width 4 as there is constraint to satisfy the separation between them. Now the wastage would be 2 as there are 2 unfilled spaces available on the table left intentionally to satisfy the constraint given in the problem.

So there would be many combinations of dish placement possibilities for a given input of different dish sizes resulting in different patterns. To solve this linear programming question, our first goal is to identify all these different patterns such that the wastage per table is minimized in any combination as much as possible. The following figure shows all the different patterns possible for the given input considering the minimized wastage of table space on that table.



Dish Type	Size	Only Hot dishes placed				Only cold dishes placed				Both
		X1	X2	X3	X4	X5	X6	X7	X8	X9
H	1	4	0	1	2	0	0	0	0	1
H	2	0	2	0	1	0	0	0	0	0
H	3	0	0	1	0	0	0	0	0	0
C	1	0	0	0	0	4	0	1	2	1
C	2	0	0	0	0	0	2	0	1	0
C	3	0	0	0	0	0	0	1	0	0
Wastage =		0	0	0	0	0	0	0	0	2

As shown above, for a given input, we get many different patterns and each pattern represents a table with particular dishes on it. Next goal would be find out the number of tables of each pattern(if possible for that pattern) in the final output such that over all wastage is minimized. Now the decision variable X_j would be the number of tables belonging to a pattern j.

X_j = number of tables belonging to a pattern j

Each decision variable from $[X_1...X_j]$ would correspond to patterns $[i...j]$. The presence of dish 1(hot, size=1) in the final output would mean that the any of the patterns $X_1, X_3, X_4 or X_9$ are also present in the output.

Therefore, the number of such dish 1 type can be given as $(4X_1 + X_3 + 2X_4 + X_9)$, which should at least be greater than or equal to given input demand(demand[0]=4) of that dish 1.

$4X_1 + X_3 + 2X_4 + X_9 \geq 4$

Similarly calculating the equations for all dishes, we get the following equations:

$2X_2 + X_4 \geq 2$
$X_3 \geq 2$
$4X_5 + X_7 + 2X_8 + X_9 \geq 6$
$2X_6 + X_8 \geq 1$
$X_7 \geq 1$

Also the non-negativity restriction which says $X_j > 0$. These are the constraints for the problem including the non-negativity restriction.

Now we need to write a objective function, the objective function here is to minimize the wastage. This is given by:

Minimize $0X_1 + 0X_2 + 0X_3 + 0X_4 + 0X_5 + 0X_6 + 0X_7 + 0X_8 + 2X_9$

The above can be written according to linear programming as:

Minimize $\sum X_j$

and the above mentioned constraints in the matrix product form as:

$\sum_j (a_{ij} * X_j) \geq b_j$

$X_j > 0$

where a_{ij} is the coefficient matrix(6x9) as there are 6 dishes and 9 patterns. And b_j is the corresponding demand array as the demand matrix(6x1).

Hence we converted the stated buffet placement problem into the following linear programming problem formulation:

Minimize $\sum X_j$

subject to

$\sum_j (a_{ij} X_j) \geq b_j$

$X_j > 0$

6.3 Solution using CPLEX OPL solver:

The above stated linear programming problem can be solved using a CPLEX solver. It is a 2 phase approach. Like we did above, the first step would be to generate all the different patterns called as pattern generation step and next step would be to pass these generated patterns as input to another model which calculates the answer for us.

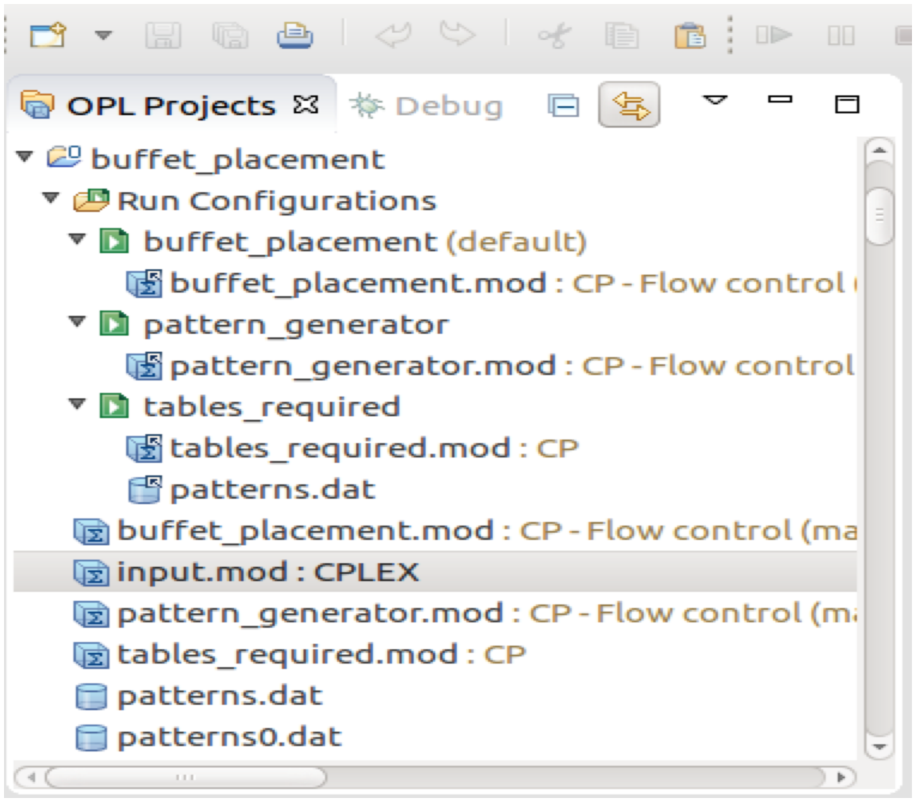
In CPLEX OPL studio, the project structure consists of Run Configurations which contain the different models(.mod) and data(.dat) files as shown below. We can create different run configurations for different models and run them for intermediary results.

My implementation consists of 3 models as follows:

pattern_generator.mod - This model generates all the patterns possible for a given input which is given in input.mod file.

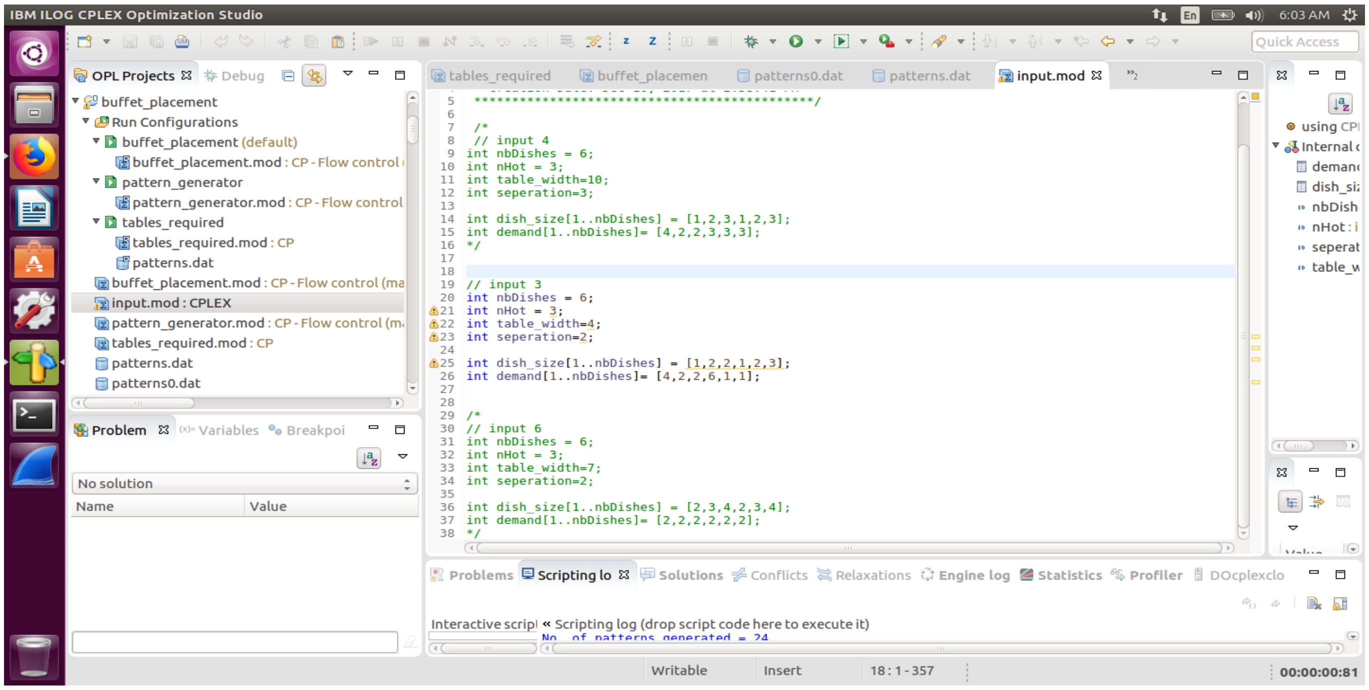
tables_required.mod - This model takes the input as the patterns from patterns.dat when user manually copies the generated patterns for a particular input and results the minimum required number of tables as output.

buffet_placement.mod - This is the main model, which runs automatically when user specifies the input in input.mod, using both the above mentioned modules and is responsible for the covering up the manual step involved. Just running this model gives us, all the generated patterns and minimum number of tables required for placing all the dishes as output.



```
buffet_placement
  Run Configurations
    buffet_placement (default)
      buffet_placement.mod : CP - Flow control
    pattern_generator
      pattern_generator.mod : CP - Flow control
    tables_required
      tables_required.mod : CP
      patterns.dat
  buffet_placement.mod : CP - Flow control (ma
  input.mod : CPLEX
  pattern_generator.mod : CP - Flow control (m
  tables_required.mod : CP
  patterns.dat
  patterns0.dat
```

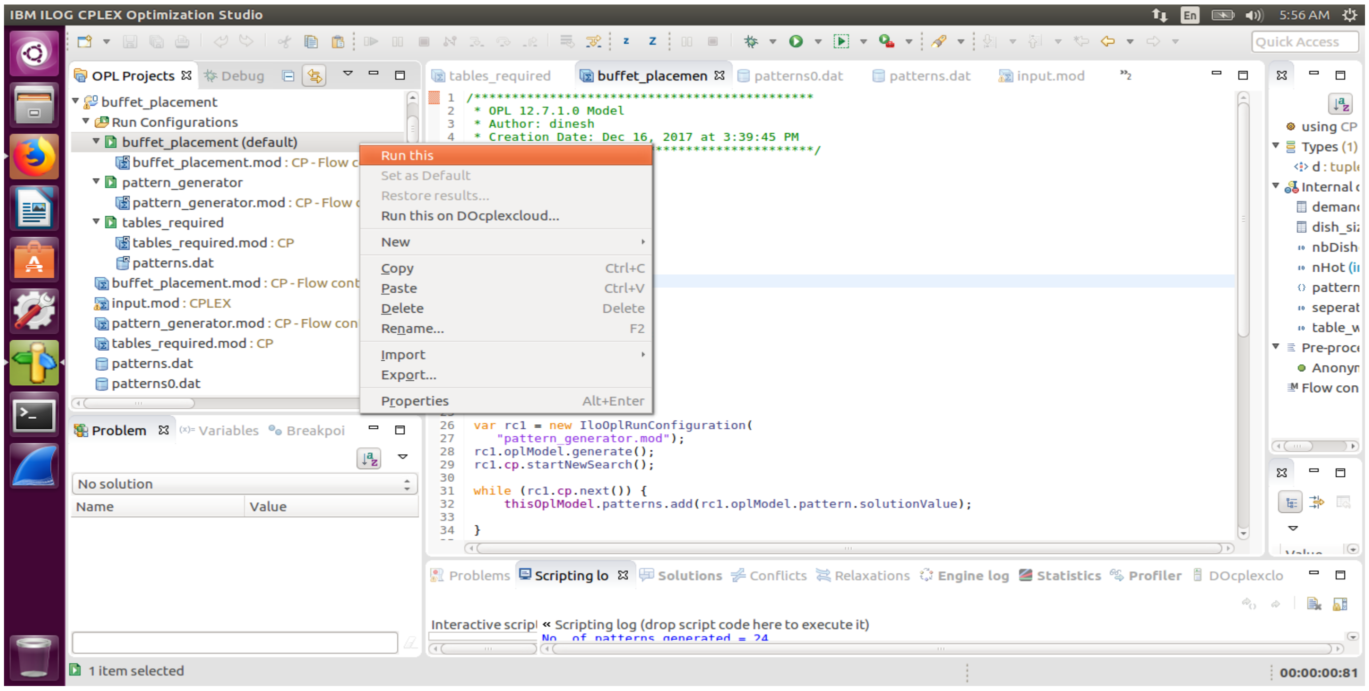
The user input can be specified in the **input.mod** file as shown below. Please find some of the sample input already present for testing purpose.



```
 **********************************************/

 /*
 // input 4
int nbDishes = 6;
int nHot = 3;
int table_width=10;
int seperation=3;

int dish_size[1..nbDishes] = [1,2,3,1,2,3];
int demand[1..nbDishes]= [4,2,2,3,3,3];
*/


// input 3
int nbDishes = 6;
int nHot = 3;
int table_width=4;
int seperation=2;

int dish_size[1..nbDishes] = [1,2,2,1,2,3];
int demand[1..nbDishes]= [4,2,2,6,1,1];


/*
// input 6
int nbDishes = 6;
int nHot = 3;
int table_width=7;
int seperation=2;

int dish_size[1..nbDishes] = [2,3,4,2,3,4];
int demand[1..nbDishes]= [2,2,2,2,2,2];
*/
```

```
No. of patterns generated = 24
```

How to run a model ? Just right click on the model as shown below and click "Run this"

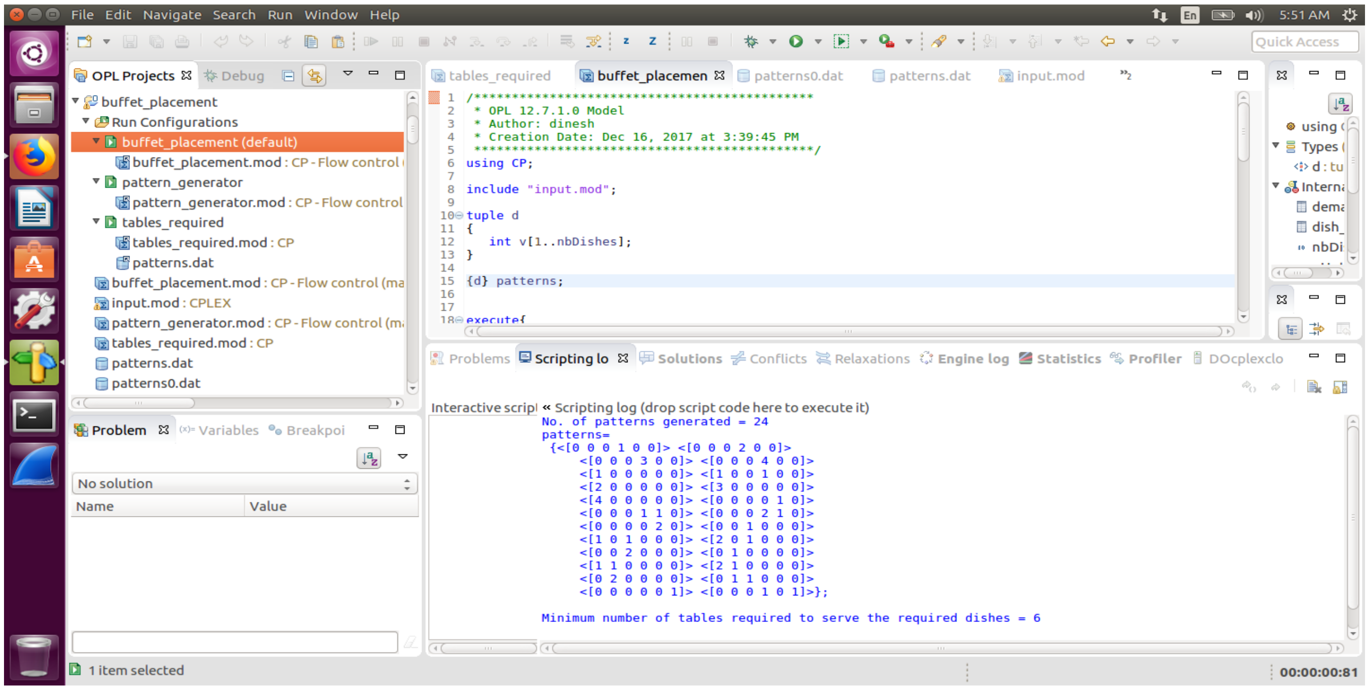


```
/********************************************
 * OPL 12.7.1.0 Model
 * Author: dinesh
 * Creation Date: Dec 16, 2017 at 3:39:45 PM
 ********************************************/
```

```
 var rc1 = new IloOplRunConfiguration(
    "pattern_generator.mod");
 rc1.oplModel.generate();
 rc1.cp.startNewSearch();

 while (rc1.cp.next()) {
     thisOplModel.patterns.add(rc1.oplModel.pattern.solutionValue);

 }
```

```
No. of patterns generated = 24
```

The output in shown in the scripting log tab as below containing the number of patters that got generated for a given input and also the required minimum number of tables to place all the dishes on the tables satisfying the given constraint of separation between hot and cold dish. All the test cases



```
/*********************************************
 * OPL 12.7.1.0 Model
 * Author: dinesh
 * Creation Date: Dec 16, 2017 at 3:39:45 PM
 *********************************************/
using CP;

include "input.mod";

tuple d
{
   int v[1..nbDishes];
}

{d} patterns;


execute{
```

Interactive scripl « Scripting log (drop script code here to execute it)

```
No. of patterns generated = 24
patterns=
 {<[0 0 0 1 0 0]> <[0 0 0 2 0 0]>
     <[0 0 0 3 0 0]> <[0 0 0 4 0 0]>
     <[1 0 0 0 0 0]> <[1 0 0 1 0 0]>
     <[2 0 0 0 0 0]> <[3 0 0 0 0 0]>
     <[4 0 0 0 0 0]> <[0 0 0 0 1 0]>
     <[0 0 0 1 1 0]> <[0 0 0 2 1 0]>
     <[0 0 0 0 2 0]> <[0 0 1 0 0 0]>
     <[1 0 1 0 0 0]> <[2 0 1 0 0 0]>
     <[0 0 2 0 0 0]> <[0 1 0 0 0 0]>
     <[1 1 0 0 0 0]> <[2 1 0 0 0 0]>
     <[0 2 0 0 0 0]> <[0 1 1 0 0 0]>
     <[0 0 0 0 0 1]> <[0 0 0 1 0 1]>};

Minimum number of tables required to serve the required dishes = 6
```

given in the homework have been run on this implementation and it had provided accurate results in all the cases. BitBucket link for the source code: ***https://bitbucket.org/dbaddam/cse505/src***

6.4 Conclusion:

Hence implemented a real life problem (buffet placement problem) in the ILOG CPLEX OPL optimization studio, using the concept of linear and constraint programming by explaining the theoretical approach of linear programming problem formulation, which is used in the paper to solve the constrained minimum break problem.

6.5 References:

1. Thanks to Prof. Dr. Fodor, for directing me to implement a real life problem in optimization language which got me hands-on experience.

2. Thanks to Prof.G.Srinivasan, Department of Management Studies, IIT Madras. Lecture Series on Fundamentals of Operations Research(http://nptel.iitm.ac.in).

3. Thanks to IBM ILOG CPLEX OPL optimization studio and student-free version download and many other references helped me to learn a new language(https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer).