REPORT - 2

220026989

1. Working of fpb2.eprime

The second file(fpb2.eprime) implements the code in a different way when compared to the main fpb.eprime. This model creates a matrix for every constraint that is linked to the solution matrix (kindly not that the b matrix here is called solution matrix inorder differentiate it from the main main fpb.eprime and to avoid confusion for the viewer/assessor). The solution matrix will not exist if any of the constraint related matrices do not exist.

An example of implementation from the program:

Constraint snippet shown(for reference): each basket should contain atleast 3 products.

```
 find each_basket_product_count_2 : matrix indexed by
[int(1..max_category), int(1..n_weeks)] of int(3..n_products)

such that

$--------------------each basket should contain atleast 3----------------
-------

    forAll i : int(1..max_category).
     forAll j :int(1..n_weeks).
      each_basket_product_count_2[i,j] = sum k : int(1..n_products).
                                          (solution[i,j,k] != 0),
$--------------------------------------------------------------------------
------
```

Here each element in each_basket_product_count_2 matrix is category x week, and each element ranges only from 3 and above. So if the sum of products for each category,each week is less than 3, the each_product_count_2 matrix will not exist, thus the solution will not exist.

2. Extra constraint added to the program

Code snippet (only for reference):

```
find equal_or_less_price : matrix indexed by [int(1..max_category)] of
int(0..max_category)
```

```
find type_matrix : matrix indexed by [int(1..max_category), int(1..n_products)]
of int(0..n_products)


such that

$-----------------------------to differentiate total quantity based on prices---
-----------

forAll i : int(1..max_category).
total_price[i] = sum k : int(1..n_products).
(category_indexes[i,k]/i)*product_quantity[k]*product_price[k],

forAll i :int(1..max_category). (total_price[i] <= basket_price[i]*n_weeks) ->
equal_or_less_price[i] = i,

forAll i :int(1..max_category). (total_price[i] > basket_price[i]*n_weeks) ->
equal_or_less_price[i] = 0,



$-------------------------------------------------------------------------------


$---------------------------extra constraint for less_or_equal price-------------

forAll i : int(1..max_category).
forAll j : int(1..n_products). (product_category[j] = i) -> type_matrix[i,j] =
product_type[j],


forAll i : int(1..max_category).
forAll j : int(1..n_products).(equal_or_less_price[i] = i /\
alldifferent_except(type_matrix[i,..], 0) /\ !same_type)  ->
((category_indexes[i,j]/i)*each_product[j] =
(category_indexes[i,j]/i)*product_quantity[j]),

$-------------------------------------------------------------
```

Here the total quantity of a specific product is multiplied with its product price and is added for each category (irrespective of the type). The total price according to the given quantity for a specific category is then compared to the basket price multiplied with the number of weeks given, of the respective category . If the calculated price is less than the basket price multiplied with the number of weeks, then I impose a constraint that the total product quantity (of a specific product,that belong to the category matched

with the above condition) should be equal to the sum of the product quantities over the weeks in the solution, given that all the types of that category is different.

This is done so as to reduce the decisions the solver has to take, to get the optimal by having the total product quantity in the solution matrix less than the given product_quantity.

3. Optimization functions implemented:

I. The basic optimisation function :
A minimal matrix of size category is constructed, wherein each element will be difference between the basket price given and the basket price we calculate according to the solution, summed out for all the weeks.
The minimal is the added to find the optimal which is minimized.

Code snippet(for reference):

```
forAll i : int(1..max_category).
minimal[i] = sum j : int(1..n_weeks) . |basket_price[i] -
price_of_each[i,j]| ,

optimal = sum i : int(1..max_category). minimal[i] * 1,
```

II. The basic optimisation function minimized over each week and then added:
A minimal matrix of size (max_category x n_weeks) where each element is the difference between basket price and total price of products in a basket (calculated from the solution) for each week. The sum of differences for each category is summed on all weeks to attain a minimal_sum matrix. The minimum value of the minimal_sum matrix is taken into small variable and then a constraint is introduced that makes each element of the minimal_sum to be less than or equal to small. After this step an optimal is calculated, as the sum of each element of minimal_sum, which is then optimized. This way, I got each of the basket price of a given category to be optimized.

Code snippet(reference only):

```
forAll i : int(1..max_category).
 forAll j : int(1..n_weeks).
 minimal[i,j] = |basket_price[i] - price_of_each[i,j]| ,
```

```
forAll i : int(1..n_weeks).
 minimal_sum[i] = sum j : int(1..max_category).  minimal[j,i] * 1,

 small = min(minimal_sum),

forAll i : int(1..n_weeks).
 minimal_sum[i] <= small,


optimal = sum k : int(1..n_weeks). minimal_sum[k] * 1,
```

4. Self review

The code works good for hard parameters when compared to the easy/medium parameters and solves with less solverNodes and Time. From the optimal values I had received through this solver code, there was an indication that some extra constraints could be implemented better so as to take care of the remaining bugs. I feel this is a whole different model from my fpb.eprime which works well for specific cases only.