NON-PARAMETRIC OPTIMIZATION APPROACH DOCUMENT

SOUMOJIT KUMAR

Introduction & Motivation:

For most retailers are faced with intricate optimization problems, regarding their decision making process, which ranges from resource allocation in space, money, and other supply chain problems, which needs attention. Among the various methods adopted, using functional forms of relationships in developing the optimization problems suffer from various drawbacks. For once, it might be intricately difficult to devise a relationship from two variables of interest and no closed form functional form be affective way of expressing the same. Even in known non-linear functional forms, addition of non-linear relations makes the optimization problem intractable and difficult to solve. Efforts of reducing these inherent complicated functions to empirical linear form reduces the accuracy and effectiveness of the model. In this respect, we devise a non-parametric method to attack such problem, which finds use cases in many scenarios. It can be interchangeably applied to space optimization, supply chain problems, resource allocation, and varied other disciplines particularly of interest to any retailer.

Promo Optimization happens to be one use case of this approach. However, our effort is to develop a generic plug-in (.jar) kind of libraries which kind of helps all kinds of such efforts. The scope of the study is expressed in Promo problem settings, but the purview is much wide and waiting to be applied to any related area. Without much ado, let us start delving into the approach.

Scope of our Study:

We attempt to devise a robust methodology which will give us a reliable and most profitable (in terms of business objectives decided) promotion planning strategy subject to promotional budget allocated for any given combination of items, department, division along with store and time dimensions. In this document, we will constraint ourselves to the optimum choice methodology of promotions picking, given an estimate of cost and lift for every promotional choice along price/store/time combination.

Methodology & elaboration:

Before getting down to the basics of our optimization approach, we summarize the entire promotional planning, as the three step method illustrated in Figure 1. In the following paragraphs we will visualize the data, explain our methodology and work out with smaller problems, and finally ending with conclusions and way forward.

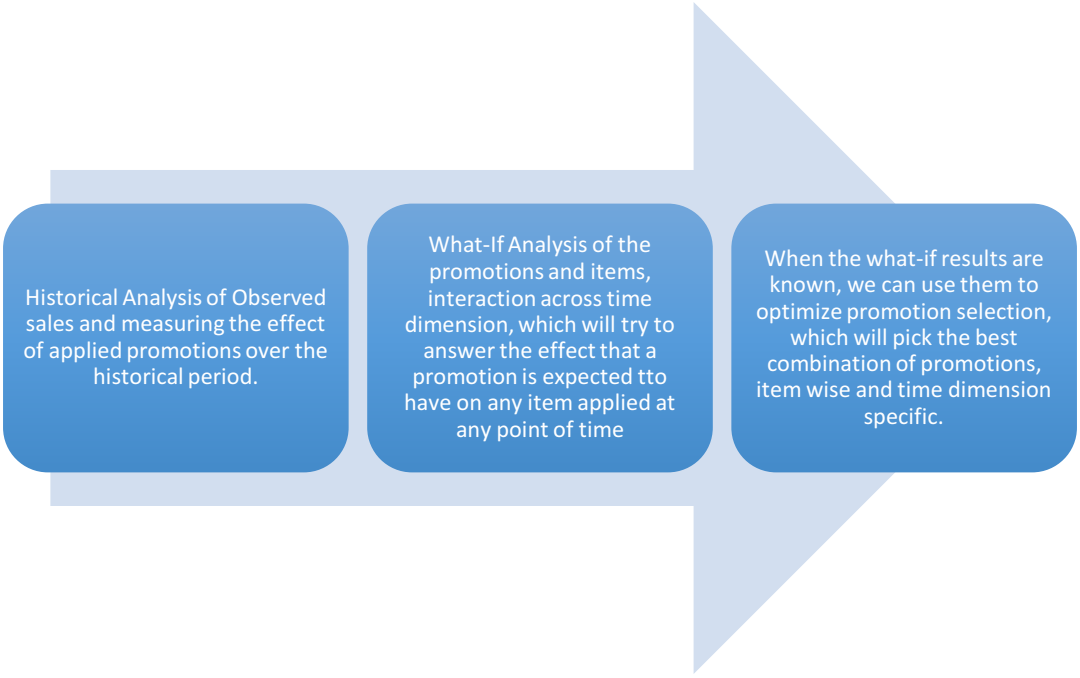

Figure 1: The above illustration exhibits the entire promotional planning cycle, from analyzing the past effects of promotions, estimating promotional planning for a future planning period and then choosing the best combination of promotions constrained within a promotion budget decided as an exogenous variable to the problem. In this study we are concentrating on the last phase of the problem, of selecting the best promotion combination, given the promotional estimates as input from the previous phases

Input Data and subsequent choice making process:

The input data as we envisage should be a table having column names as "Stores", "Items", "Promotion-vehicle and promotion combination", "Time" as the attributes of any row. In addition, to that each row, will have "Estimated Lift" and "Dmarkdown Dollars" as additional rows, which is an output of the previous stages of the promotion problem cycle as explained earlier.

Time(T)	***Stores(S)***	***Items(I)***	***Promotion combination with promo vehicle(P)***	***Estimated Lift(L)***	***Dmarkdown Dollars(C)***
Normally between 1-52 resembling 52 week of a year. Can be anything within the planned period.	All the stores on which planning is done	Item Ids, which may be within a particular division, department or the entire item set.	Promotion combinations which are likely to be applied along with their promo vehicle	A revenue number which suggests the increase of revenue on application of the promotion	Cost incurred in running the promotion

Given the possible number of objects in each attribute i.e. *T, S, I* or *P*, we can be looking at a very large table of *mod(T)*mod(S)*mod(I)*mod(P)*, among which we

need to choose those rows, summation of whose C does not exceed a promotional budget B, and the sum of L is maximized. However, we can try to reduce the size of problem by tackling subsets of *T, S, I* or *P*, in turn, instead of attacking the entire universe of maximum number of combinations. In that case, we need to break B into smaller sizes, as the choice of smaller universe suits.

Worked out small examples as illustration

We can think of some ways to tackle the problem: a) Branch & Bound, b) Dynamic Programming or c) Recursion. We will explain each of them by solving a small representative example. We have also implemented the Branch and Bound on a set of representative public data for experimental results. In this report, we will also attach our results.

Let us suppose, we are tackling 3 combinations having value and weights given, analogous to our problem, where weight signifies cost and value corresponds to the lift. We will also have a capacity constraint (K) resembling promotional budget in ur original mother problem. Given these data, we tread to find the best choice among the 3 items. It is noteworthy to mention here, that a straight forward brute force method will result in a search space of 2^3 combinations, which may be feasible in a small scenario, but when we extend to a generalized case of N items, having 2^N combinations to be examined, we are doomed to fail using a brute force method.

i	V_i	W_i
1	45	5
2	48	8
3	35	3

Example 1: The table resembles the data, where we have 3 items to be chosen in a capacity K of 10. Each of the items corresponding value and weight are given in V, W columns along with the items. In the following paragraphs we extend the B&B methodology to the example

Please consider the below diagrams for following the steps of proposed B&B methods. Two methods are considered; the latter is faster than the former. The second one differs in calculation of upper bounds and hence forms a tighter upper bound, and cuts branches much earlier. This makes it possible to traverse or visit lesser number of nodes in the pursuit of the optimal, making it faster. We will also attach the code link, which has been implemented in Java, and the representative data sets on which it is experimentally run. ***Please note that the experimental runs are for full search time, giving a guaranteed optimal. For all practical purposes, we can converge to a very good solution nearby to the guaranteed optimal in a pretty reasonable time. For experimentation purpose, we are checking the limits of full optimality search time. However, that might not be necessary in practical implementation runs.***

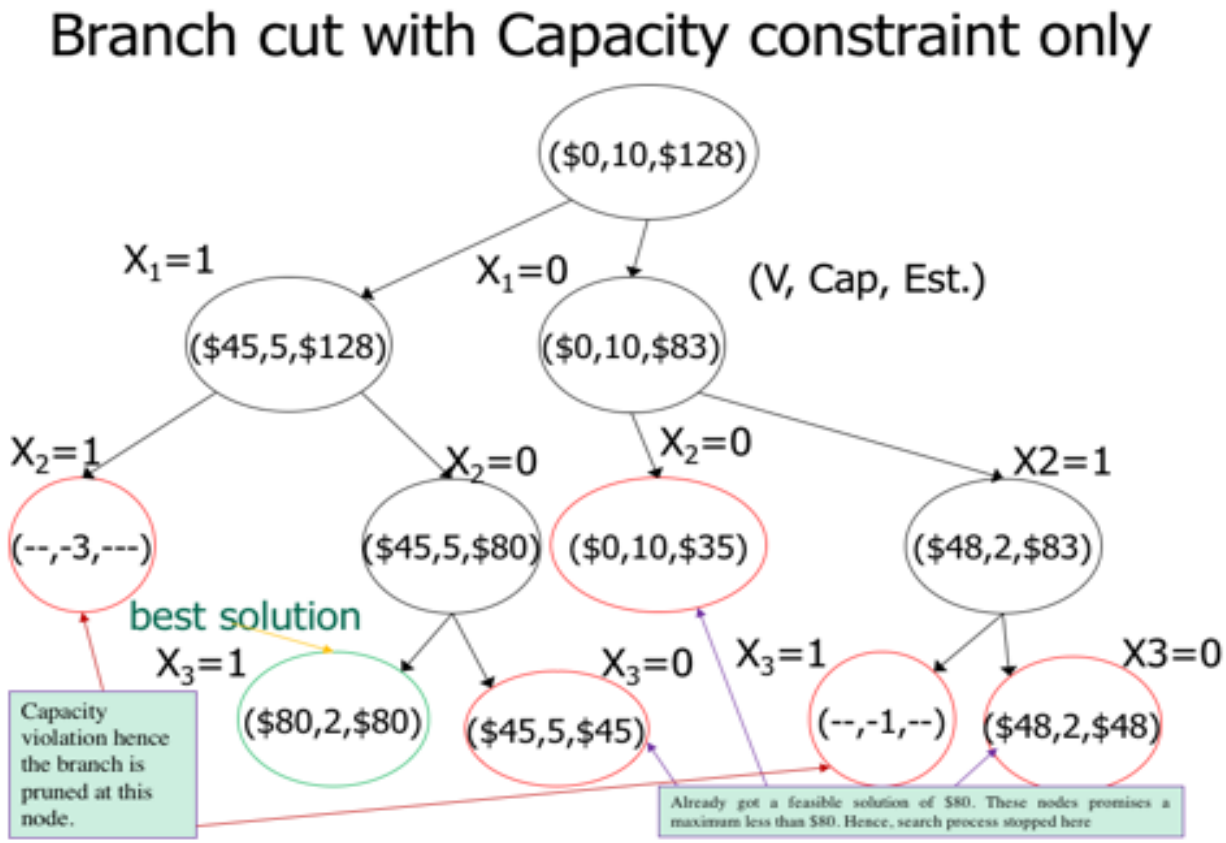

Figure 2: It illustrates the full branch and bound computation for the example problem. It starts with a root node completely empty, in terms of items being chosen in the optimal set. At every node, the algorithm takes a decision whether or not, to consider the next item in the optimal set and breaks into corresponding child nodes. At the child node, depending on whether the item is chosen or not, the available capacity is reduced by the weight of the item chosen. The available value of the node increases to the value of the item chosen.

In each of the nodes we are using a Triplet to resemble the state of the node as (V, Capt., E). "V" is the available value accumulated to the node, by choosing the items along the path to the node from the root. Similarly, "Capt." is the left over capacity remaining on reaching that node. It is calculated by reducing the sum of the weights of the chosen items along the path from the root to the node from K. "E" is the estimated upper bound estimate of the maximum value that can be made from the node, given prior decisions in the path from the root to the node are already made. "E" is calculated by adding all the item values which are still not in the decision set from the node, signifying, in an unconstrained scenario, all the left over items can be chosen as a best case, and hence forms an upper bound. We consider capacity constraint and upper bound as the pruning method. Whenever, selecting additional item surpasses K, we prune the branch and search stops on that branch. Similarly, if we get a feasible solution whose value is more than the estimated upper bound of value could be achieved we also forgo any path below the node and prune it. Please note that at each decision node Xi are the decision variables which if 1, the item 1 is chosen and not chosen when the variable is 0. At every decision point we are facing with the choice of selecting any particular item and the tree grows until prune all branches, or reach leaves, and choose the maximum V among all leftover feasible leaves.

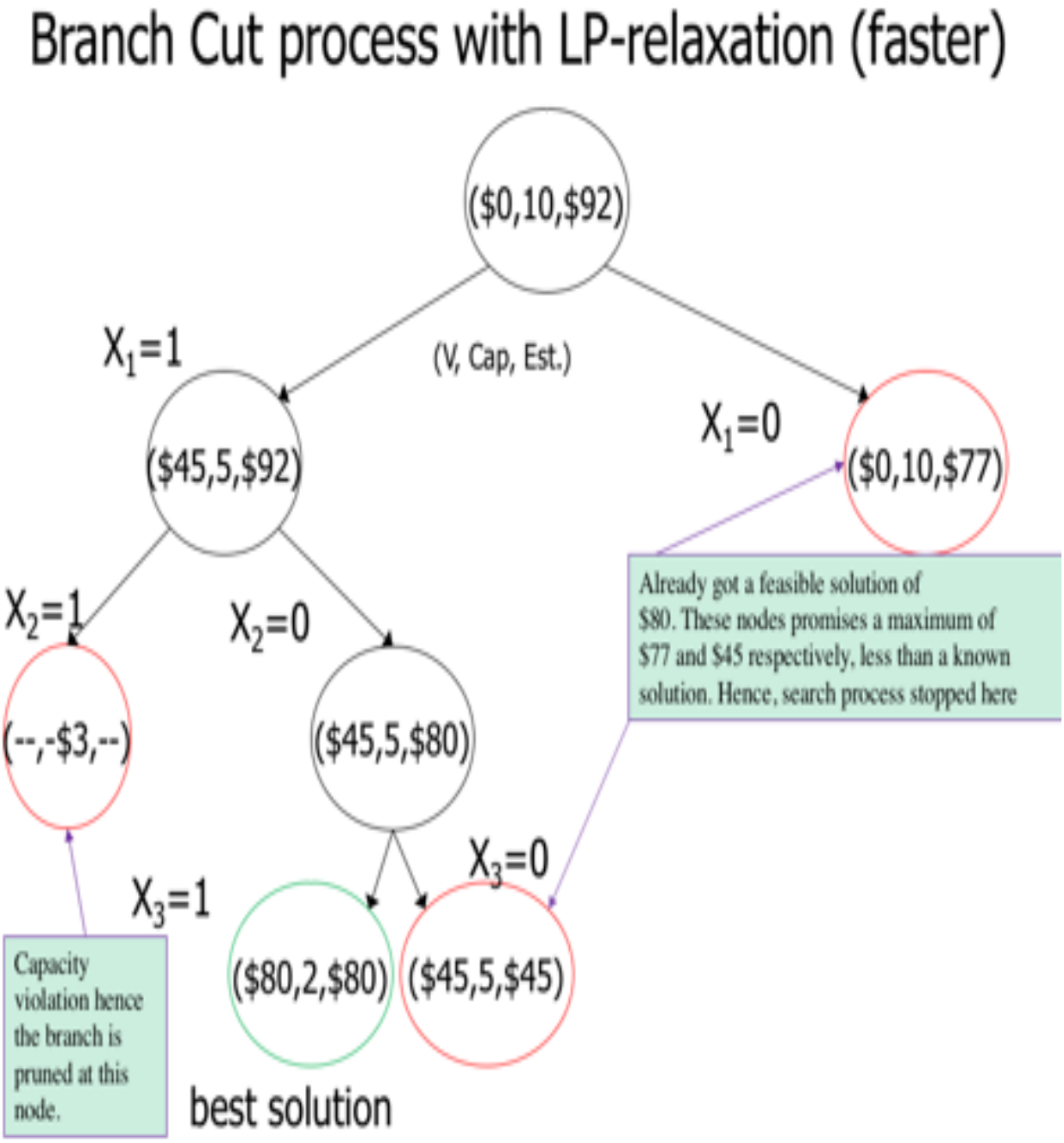

Figure 3: The B&B second method, where all the principles remain same above, except the calculation of "E" method. At each node, we use Fractional Knapsack methodology to calculate the upper bound of maximum possible value that can be reached from that node. See a complete guide on Fractional or Continuous Knapsack Problem at https://en.wikipedia.org/wiki/Continuous_knapsack_problem

Code and Input Data

The Code is uploaded at Git at https://git.target.com/Z001YS6/Knapsack. The experimental data set is loaded at **data** folder in the repository. It contains data from about 4 items up to 10000 items. 18 files are there of varied items, as stated, for 18 separate runs with different loads. We tried to run all of those to full optimality to find the maximum time taken for full completion. ***However, we reiterate, that full search and guaranteed optimality might not be necessary in actual runs, and we can reach optimal or very near-optimal results even for very large input size in a matter or seconds or minutes. The bigger problem is to solve the "Stackoverflow" error for very large input size.***

How to read input from the data files and output from the code run

The code input data folder contains $n + 1$ lines. The first line contains two integers, the first is the number of items in the problem, n. The second number is the capacity of the knapsack, K. The remaining lines present the data for each of the items. Each line, $i \in 0, 1 \ldots, n - 1$ contains two integers, the item's value v_i followed by its weight w_i.

Input Format

```
n K
v_0 w_0
v_1 w_1
...
v_n-1 w_n-1
```

The output prints the objective value and the next line is a list of n 0/1-values, one for each of the x_i variables, signifying whether i item is selected or not. Other than that, lot of messages are popped up to signify the different points of computation. However, final two lines give the objective and the choice of items. It also gives the time run for full guaranteed optimal solution.

Time Experiments run with the 18 input files

We ran the 18 input files, each to full optimality. **Again, we state that this full optimality search is for academic/experimentation purpose only, and in original practical runs, we need not go full extent of guaranteed optimal solution**.

The time results are tabulated below, which points out certain interesting points. It may be noted that some inputs of 200, 300 and 400 are taking very large amounts of time, even in comparison to higher input size as 500 and 1000. Such aberrations

are also seen in upper rows. It points out that apart from input size there are various factors which contribute to the guaranteed optimality search. *It is how quickly, the branches are pruned off, to complete tree search. It might be the case, that higher input sizes files had their branches pruned quickly to the amount that the search space became much smaller, even though the potential search space might be much larger at the start of the computation in comparison to lower sized input.*
It points us in thinking that search time depends on,

1. *Order of the traversing of branches.*
2. *K*
3. *Individual data points of the items*

Hence, it becomes clear that we can built more intelligent heuristics like local neighborhood search, smart tree traversal methods, and tighter upper bound calculation at each node, which will prune nodes much faster.

File	Items to be considered	Time Taken(Seconds)
ks_4_0	4	0.027 seconds
ks_19_0	19	0.05 seconds
ks_30_0	30	0.181 seconds
ks_40_0	40	57.514 seconds
ks_45_0	45	1.416 seconds
ks_50_0	50	0.92 seconds
ks_50_1	50	1.72 seconds
ks_60_0	60	153.67 seconds
ks_100_0	100	3586.87 seconds (59.78 minutes)
ks_100_1	100	63.535 seconds
ks_100_2	100	7.852 seconds
ks_200_0	200	56285.33 seconds (15.63 hours)
ks_300_0	300	>> 24 hours
ks_400_0	400	>>24 hours
ks_500_0	500	378.533 seconds (6.31 minutes)
ks_1000_0	1000	891.47 seconds (14.86 minutes)
ks_10000_0	10000	Stack Overflow error-- could not be run on MAC OS X, 16 GB local machine

Other Approaches

Among other approaches, we have discussed Dynamic Programming and Recursive method. We will add an example of Dynamic Programming Approach. However, we feel that implementation of the second B&B approach would be best for our business needs, as it gives the flexibility of adding constraints as checks. The failure

of any constraint will prune the branch.

Let us work out a Dynamic Programming small example.

i	V_i	W_i
1	16	2
2	19	3
3	23	4
4	28	5

Example 2: The table resembles the data, where we have 4 items to be chosen in a capacity K of 7. Each of the items corresponding value and weight are given in V, W columns along with the items. In the following paragraphs we extend the Dynamic programming methodology to the example

Capacity	**0**	**1**	**2**	**3**	**4**
0	0	0	0	0	0
1	0	0	0	0	0
2	0	16	16	16	16
3	0	16	19	19	19
4	0	16	19	23	23
5	0	16	35	35	35
6	0	16	35	39	39
7	0	16	35	42	**44**

In this method, we draw a table having the discretized capacity as rows and number of items as columns. Here, capacity given is 7, and hence, 8 rows are created {0,1, … ,7} to resemble the capacity states at those values. Similarly, as 4 items are there, 5 columns are created {0,1, …, 4}. Here, at each cell $[i, j]$ the optimal possible value is created, as if, there are j items and i capacity. At each node $[i, j]$, we check the left side node $[i, j-1]$ and check the node at $[i - w_j, j-1]$, where w_j is the weight of the j item. Then we put the value at the cell as,

$$value[i,j] = \max\left(value[i,j-1], w_j + value\left[i - w_j, j-1\right]\right)$$

By this way, we complete the entire table, and by the lowermost, rightmost corner cell, we arrive at the solution of our example problem, and get a value of 44. This method, is however, very space constraining, and might be difficult to implement, when other constraints are also added. But, nevertheless, it gives an elegant method for solving similar family of problems.

In the next method, we give a pseudo-code for recursive approach to the problem. This is also another thought process, which if, not used immediately in this endeavor, might provide handy in future scenarios. Let us suppose $O\ (int\ K, int\ j)$ be the function, where K is total space as used earlier, j is the list of items having weight w_i

And value v_i for any of its i item ($i \in j$).

$$
\begin{aligned}
&int\ O\ (int\ k, int\ j)\{\\
&\quad if(j == 0)\\
&\qquad return\ 0;\\
&\quad else\ if\ \left(w_j \leq K\right)\\
&\qquad return\ \max\ (O\ (K, j-1), v_j + O\left(K - w_j, j-1\right))\\
&\qquad else\\
&return\ O(k, j-1)\}
\end{aligned}
$$

Conclusions:

In all, we conclude that we are getting very encouraging results to continue our endeavor to built a full scale module with the approaches we have discussed here. We feel, these methodologies can be scaled up with Data Engineering help, and if given proper support, we can proceed with this development.