UNIVERSITÀ DEGLI STUDI DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

Implementation of The Multiple Knapsack Problem in the Kernel Search

Teacher:

Ch.ma Prof.sa Renata Mansini

Students:

Matteo Beatrice (739848)

Luca Cotti (719204)

Giacomo Golino (719210)

Academic Year 2021/2022

Contents

Introduction

In this report we describe the process of adapting the Kernel Search, a generic heuristic framework that can be successfully used on a variety of problems, to the specific model of the Multiple Knapsack Problem.
While Kernel Search provides good solutions in a short amount of time, tuning for a specifi problem becomes necessary to improve the performance of the algorithm and the quality of the solution found.

In the first chapter, the Multiple Knapsack Problem is introduced, along with a few real life examples to better explain the model.
Chapter 2 contains a brief description of the Kernel Search algorithm, along with a few basic improvements to the default algorithm.
In chapter 3 we describe the technical specifications of our implementation of the Kernel Search algorithm, while in chapter 4 we study the performance of the implementation and the test instances used as benchmark.
In chapter 5 we detail all the methods that we have introduced to better adapt the Kernel Search algorithm to the Multiple Knapsack Problem.
Finally, in chapter 6 we report the results of the testing we have executed on the modified Kernel Search algorithm.

1. The Multiple Knapsack Problem

The Multiple Knapsack Problem (MKP) is a strongly NP-hard problem that was described in [1] as follows: given a set of m knapsacks with a known capacity c_i ($i = 1, \ldots, m$) and a set of n items with known profit p_j and weight w_j ($j = 1, \ldots, n$) the MKP consists in selecting m disjoint subsets of items (one subset per knapsack) such that the total weight of the items in the knapsack does not exceed its capacity, and the profit of the selected items is maximized.

1.1 Mathematical Model

There are quite a few formulations for the MKP, with different degrees of complexity and performance.

The model that was used in this project is the classical and most intuitive one, that uses the following binary variables:

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is packed into knapsack } i; \\ 0 & \text{otherwise.} \end{cases}$$

The mathematical formulation is the following:

$$\max z = \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \tag{1.1}$$

$$\sum_{j=1}^{n} w_j x_{ij} \leq c_i \qquad i = 1, \ldots, m \tag{1.2}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad j = 1, \ldots, n \tag{1.3}$$

$$x_{ij} \in \{0, 1\} \qquad i = 1, \ldots, m \quad j = 1, \ldots, n \tag{1.4}$$

The objective function (1.1) maximizes the profit of the items in the knapsacks.

Constraints (1.2) impose that the capacity of each knapsack is respected, while constraints (1.3) ensure that each item is packed in no more than one knapsack.

We can assume that each knapsack can contain at least one item ($\min_j\{w_j\} \leq \min_i\{c_i\}$) and that each item can fit in at least one knapsack ($\max_j\{w_j\} \leq \max_i\{c_i\}$).

1.2 Real Life Examples

Various real life examples of the use of MKP can be made.

One of the most straightforward ones is the *cargo organization* for a transport company. In this scenario, the knapsacks are the trucks or containers. Their capacity is the effective capacity of the truck/container.
The items are the objects that must be delivered: their weight is the actual weight (or volume) of the goods that need to be shipped, while the profit is the priority of the delivery.

Another example is the *project selection*: the knapsacks are the years available for the development of projects, and their capacity is the available budget for that year.
The items are the available projects: the weight is the cost of the project, the profit is the monetary gain from the project completion.

More complex examples can be found in the literature. Applications in the design of computer processors, layout of electronic circuits and sugar cane alcohol production can be found in [2]. Uses in vehicle and container loading are mentioned in [3].

2. Kernel Search

Kernel Search is a heuristic framework used to solve MIP problems, which was introduced in [4]. The algorithm has been studied in a few applications, including the *Multidimensional Knapsack Problem* in [5] and the *Portfolio Selection Problem* in [6].

The algorithm is based on a few observations that can often be made when solving MIP problems:

- In the optimal solution there are only a few non-zero variables;
- Basic variables in the LP-relaxation are good predictors of non-zero variables in the MIP optimum;
- Reduced costs are good predictors of the likelihood of a non-basic variable to be in the MIP optimum.

Kernel search needs an MIP solver such as GUROBI or CPLEX to solve a set of MIP sub-problems.

2.1 Algorithm

The algorithm has two main phases: an *initialization phase* that builds an initial *kernel set* and a number of *buckets*, and an *improvement phase* that iteratively enlarges the kernel set and improves the solution.

2.1.1 Initialization Phase

The first step of the Kernel Search is to solve the LP relaxation of the problem.
Then, the set of variables of the model is sorted according to some *sorting criterion*.

Once this is done, the kernel set Λ is built by selecting the C variables in the sorted set according a *kernel construction criterion*, where C is the size of the kernel set.

The next step is to solve MIP(Λ), which is the original problem restricted to only include the variables in Λ: all the variables not in Λ are set to 0 to exclude them from the problem.
The resulting solution will be called x^* and the optimum value w^*.
It is possible that no solution is found for MIP(Λ): in this case w^* is set to $-\infty$.

The variables not in Λ are partitioned into a certain number of buckets nb, according to a *bucket construction criterion*.

2.1.2 Improvement Phase

The main objectives of this phase are finding a new improving feasible solution and identifying new variables to enter the kernel set.

Once the buckets are constructed, the algorithms proceeds with the improvement phase: for each bucket B_i $(i = 1, \dots, nb)$, the restricted sub-problem MIP($\Lambda \cup B_i$) is solved.

Only the feasible sub-problems with an incumbent solution $\bar{x^*} \geq w^*$ are considered.
For such sub-problems, the set $\bar{\Lambda}_i \coloneqq \{\text{selected variables in } B_i\}$ is built.
Then the kernel set is updated: $\Lambda \coloneqq \Lambda \cup \bar{\Lambda}_i$. Since variable are only added to the kernel, and never removed, its size increases monotonically.

After all the nb have been iterated, the algorithm ends.

2.2 Parameters

The Kernel Search has quite a few configuration parameters that can be used to tune the algorithm for a specific problem:

- *Variable sorting criterion*: used to sort the variables in the initialization phase. Ideally, if the sorting criterion is selected appropriately, all the significant variables for the problem should be located in Λ and the very first buckets.
- *Kernel size C*: the number of items initially selected to construct the kernel.
- *Kernel construction criterion*: used to build the kernel. The criterion defined in the basic Kernel Search consists in selecting the first C variables in the sorted set, but for certain problems more complex criteria can be used.
- *Bucket construction criterion*: how the buckets are constructed. Also defines the number and size of the buckets.

2.3 Improving Efficiency of the Basic Kernel Search

To improve the performance of the basic Kernel Search, two constraints can be added when solving each MIP($\Lambda \cup B_i$):

$$\text{value of the objective function} \geq w^* \tag{2.1}$$

$$\sum_{j \in B_i} x_j \geq 1 \tag{2.2}$$

The *cutoff constraint* (2.1) allows only solutions that improve on the current objective value. Constraints (2.2) ensure that at least one item must be selected from the bucket B_i: such an item will then be included in the new kernel set.

2.4 Iterative Kernel Search

Iterative Kernel Search is a variation of the basic Kernel Search where buckets are scrolled more than once. The algorithm is as follows:

1. Execute the basic Kernel Search;
2. Set $nb \coloneqq q - 1$, where $q \coloneqq \max\{i : \bar{\Lambda}_i \neq \emptyset\}$;
3. Execute the improvement phase of the algorithm;
4. If a new variable enters the kernel set, *nb* is reset to its initial value, the improvement phase is executed again, and the algorithm is repeated from step 2.

A simpler version of this algorithm repeats step 3 a certain number of times, skipping steps 2 and 4 entirely. The *number of iterations* is a configuration parameter.

3. Implementation

The Java source code for the Kernel Search was provided during the course, and it implements a simple iterative Kernel Search (as explained in 2.4), using GUROBI as the MIP solver. It also includes the improvements described in 2.3.

The code, which can be found alongside its documentation at `https://github.com/Golino98/KernelSearchGolinoCottiBeatrice`, was initially modified to implement the MKP model described in 1.1, and then refactored to improve the code quality and efficiency.
The algorithm itself and the configuration parameters were not changed at this time.

The Kernel Search is configured with the following parameters:

- *Variable sorting criterion*: sort variables by non-increasing value and non-decreasing reduced cost.
- *Kernel size* C: 15% of the number of variables (rounded to the nearest integer).
- *Bucket construction criterion*: iterates through the sorted variables, grouping them in buckets of size equal to the 2.5% of the number of variables (rounded to the nearest integer). It's possible for the last bucket to contain fewer items than the previous ones, because the number of remaining variables could be inferior to the bucket size.
- *Number of iterations*: 2.

3.1 GUROBI

The project uses GUROBI 9.5, which is the latest version of GUROBI available at the time of writing.
The GUROBI configuration is the following:

- *Presolve*: 2 (aggressive presolve)
- *MIPGap*: 1×10^{-12}
- *Threads*: 12

4. Computational Experiments

In this section we report the results of tests executed on the default Kernel Search applied to the MKP. The main objective of these experiments is to determine the performance of the default Kernel Search, both in terms of quality of the solution found and time required for the execution.

4.1 Test Instances

The test instances for the MKP were kindly provided to us by the authors of [1].
There are in total 2100 test instances, organized in five directories of increasing complexity:

- SMALL
- FK_1
- FK_2
- FK_3
- FK_4

SMALL contains 180 instances with $m \in 10, 20$ and $n \in 20, 40, 60$, while FK_1, FK_2, FK_3, FK_4 contain 480 instances each, designed with the aim of identifying critical ratios of n/m that produce difficult instances.

The weights w_j are always uniformly distributed in $[\alpha, 1000]$, with $\alpha = 1$ for the SMALL instances and $\alpha = 10$ for the FK instances.

Each instance belongs to one of the following four correlation classes:

- *uncorrelated*: profits p_j are uniformly distributed in $[\alpha, 1000]$;
- *weakly correlated*: For SMALL, $p_j = 0.6w_j + \theta_j$, with θ_j uniformly random in $[1, 400]$. For FK, the p_j values are uniformly distributed in $[w_j - 100, w_j + 100]$, such that $p_j \geq 1$;
- *strongly correlated*: For SMALL, $p_j = w_j + 200$, for FK, $p_j = w_j + 10$;
- *subset-sum*: $p_j = w_j$.

All the instances in SMALL and FK_1 were already solved to the optimum. Of those in the three remaining folders only a few were already solved: the unsolved ones only had an upper and lower bound.

4.1.1 Format of the Instances

The instances are contained in plain `.txt` files.

An example of the structure of an instance is shown below:

$$\left.\begin{array}{l}3\end{array}\right\} \rightarrow \text{integer that indicates the value of } m \textit{ (number of knapsacks).}$$

$$\left.\begin{array}{l}7\end{array}\right\} \rightarrow \text{integer that indicates the value of } n \textit{ (number of items).}$$

$$\left.\begin{array}{l}908\\ 834\\ 675\end{array}\right\} \rightarrow \text{integers that indicate the capacity } c_i \text{ of the knapsack i with } i = 1, \ldots, m.$$

$$\left.\begin{array}{ll}264 & 430\\ 606 & 945\\ 268 & 409\\ 619 & 591\\ 958 & 839\\ 972 & 818\\ 723 & 71\end{array}\right\} \rightarrow \left.\begin{array}{l}\text{First column shows the weight } w_i\\ \text{Second column shows the profit } p_i.\end{array}\right\} \rightarrow i = 1, \ldots, n.$$

The execution times required to solve the instances to the optimum were not provided to us. For a few of the instances we were able to find the exact solution using a GUROBI solver with the following configuration:

- *Presolve*: 2 (aggressive presolve)
- *Time limit*: 1 hour
- *Threads*: 12
- *MIPGap*: 1×10^{-12}

Most of the instances however used all the available time without reaching the optimum. The code and the results can be found at `https://github.com/Golino98/EsattoMKP`.

4.1.2 Selected Instances

To simplify the testing of the changes to the Kernel Search algorithm, we only considered 40 instances: 6 solved to the optimum for each directory, plus 10 that were not solved to the optimum.

Tables 4.1 and 4.2 represent, respectively, the optimum values of the 30 instances solved to the optimum, and the upper and lower bound for the 10 that couldn't be solved.

4.2 Performance of the default Kernel Search

Table 4.3 outlines the result of tests executed on the Kernel Search algorithm with the default configuration, as outlined in 3. The tables include, for each instance:

- Directory of the instance;
- Name of the instance;
- Solution found;
- Time elapsed;
- A boolean value that specifies if the time limit (120s) was reached.

Directory	Instance	OPT
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15534
	probT1_0U_R50_T002_M020_N0060_seed01	26593
	probT1_1W_R50_T002_M010_N0040_seed09	12724
	probT1_1W_R50_T002_M010_N0060_seed10	19652
	probT1_1W_R50_T002_M020_N0020_seed10	3429
	probT1_1W_R50_T002_M020_N0040_seed10	12405
FK_1	random10_60_1_1000_1_12	23064
	random10_100_2_1000_1_10	29800
	random12_48_3_1000_1_16	11865
	random15_45_1_1000_1_13	15160
	random15_75_3_1000_1_16	18321
	random30_60_4_1000_1_14	11017
FK_2	random20_120_1_1000_1_12	47823
	random20_200_2_1000_1_14	53618
	random24_96_3_1000_1_18	23912
	random30_90_4_1000_1_20	26038
	random30_150_2_1000_1_17	44518
	random60_120_4_1000_1_19	20463
FK_3	random30_180_1_1000_1_20	75618
	random30_300_2_1000_1_4	85147
	random36_144_3_1000_1_17	39245
	random45_135_4_1000_1_16	37005
	random45_225_3_1000_1_15	56199
	random90_180_4_1000_1_19	30047
FK_4	random50_300_1_1000_1_16	119973
	random50_500_2_1000_1_14	135583
	random60_240_3_1000_1_14	61180
	random75_225_4_1000_1_17	60458
	random75_375_2_1000_1_5	101917
	random150_300_2_1000_1_12	55707

Table 4.1: Exact solutions for the 30 instances already solved to the optimum

Directory	Instance	LB	UB
FK_3	random30_180_2_1000_1_2	47911	47966
	random36_144_2_1000_1_20	42988	43004
	random45_135_3_1000_1_12	37680	37681
	random45_135_4_1000_1_14	52666	33756
	random45_225_2_1000_1_20	68070	68125
FK_4	random50_300_2_1000_1_5	80459	80483
	random60_240_1_1000_1_17	95602	95792
	random60_240_2_1000_1_15	66497	66529
	random60_240_3_1000_1_18	59637	59648
	random75_225_4_1000_1_20	58338	58340

Table 4.2: Upper and lower bounds for the 10 instances not solved to the optimum

Directory	Instance	OPT	Time Elapsed	Time limit Reached
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15440	118.6632536	true
	probT1_0U_R50_T002_M020_N0060_seed01	25636	19.3855657	false
	probT1_1W_R50_T002_M010_N0040_seed09	12699	9.6504811	false
	probT1_1W_R50_T002_M010_N0060_seed10	19610	118.7947297	true
	probT1_1W_R50_T002_M020_N0020_seed10	3429	4.5942635	false
	probT1_1W_R50_T002_M020_N0040_seed10	12233	13.6375786	false
FK_1	random10_100_2_1000_1_10	29714	120.9639609	true
	random10_60_1_1000_1_12	23064	58.006012	false
	random12_48_3_1000_1_16	11837	38.1244613	false
	random15_45_1_1000_1_13	14726	10.0776358	false
	random15_75_3_1000_1_16	18296	120.3020247	true
	random30_60_4_1000_1_14	11017	37.2557905	false
FK_2	random20_120_1_1000_1_12	47785	120.7477216	true
	random20_200_2_1000_1_14	53329	120.1514607	true
	random24_96_3_1000_1_18	23848	120.3337222	true
	random30_150_2_1000_1_17	44176	118.2145149	true
	random30_90_4_1000_1_20	26021	102.1200707	false
	random60_120_4_1000_1_19	20463	118.029522	true
FK_3\notSolved	random30_180_2_1000_1_2	47633	120.7227232	true
	random36_144_2_1000_1_20	42594	118.0888587	true
	random45_135_3_1000_1_12	37558	118.1862276	true
	random45_135_4_1000_1_14	33709	120.5998127	true
	random45_225_2_1000_1_20	67629	118.0391075	true
FK_3\Solved	random30_180_1_1000_1_20	75293	120.7290469	true
	random30_300_2_1000_1_4	84727	121.2111745	true
	random36_144_3_1000_1_17	39128	118.5823754	true
	random45_135_4_1000_1_16	36959	120.8276246	true
	random45_225_3_1000_1_15	55933	121.3279636	true
	random90_180_4_1000_1_19	30047	118.3908919	true
FK_4\notSolved	random50_300_2_1000_1_5	79848	120.8818098	true
	random60_240_1_1000_1_17	95156	121.2438769	true
	random60_240_2_1000_1_15	65765	121.4972109	true
	random60_240_3_1000_1_18	59315	120.2022617	true
	random75_225_4_1000_1_20	58027	122.210206	true
FK_4\Solved	random150_300_2_1000_1_12	55101	122.4559536	true
	random50_300_1_1000_1_16	119327	120.7342642	true
	random50_500_2_1000_1_14	134389	118.8051819	true
	random60_240_3_1000_1_14	60817	118.2096069	true
	random75_225_4_1000_1_17	55073	121.7853344	true
	random75_375_2_1000_1_5	100595	123.0345909	true

Table 4.3: Results of Kernel builder percentage with variables sorted by value and absolute RC

5. Improvements

In this section we will describe in chronological order the various changes that we applied to the default Kernel Search implementation, with the objective of improving its performance in solving the MKP instances.

For the sake of completeness, we will describe all attempts at improving the algorithm, including the unsuccessful ones.

5.1 Variable Sorting

As outlined in [5] and [6] the quality of the solutions found by Kernel Search heavily depends on the criterion used to sort the variables.

In [5], a sorting criterion for the *Multidimensional Knapsack Problem* is mentioned: sorting by non-increasing *LP relaxation value* and non-decreasing *absolute value of the reduced cost.* This is the default criterion that was provided in the code, and the experiments we executed on the instances proved that this sorting also provides high quality solutions for the MKP.

Other criteria we have tested are:

- Sort by non-decreasing *reduced cost*, breaking ties using the non-increasing LP relaxation value;
- Sort by non-increasing *ratio of profit and weight of the item*, breaking ties using the non-increasing *reduced cost value*;
- Sort by non-increasing *value of the LP relaxation* multiplied by the *ratio of profit and weight*, breaking ties using the non-decreasing *reduced cost*;
- Sort randomly.

Generally, the efficiency of a sorting criterion depends on the kernel construction criterion used.

5.2 Kernel Construction Criterion

A recurring problem we have identified when running complex instances (such as the one in FK_4) is that, from the very beginning, the kernel contains quite a lot of variables, which makes the solving of the kernel problem and the buckets sub-problems slow.

This means that a starting point to improve the performance of the Kernel Search could be to optimize the management of the kernel set, possibly starting from the *kernel construction criterion*, which by default is to include the first C sorted variables (Percentage kernel).

We have experimented with the following criteria:

1. Positive kernel: select the variables that have a value greater than 0 in the LP relaxation;
2. Integer kernel: select the variables that have a value equal to 1 in the LP relaxation;
3. Threshold kernel: select the variables that have a LP relaxation value greater than a threshold, that we have set to 0.6.

5.3 Overlapping Buckets

A criterion that could improve the quality of the solution of the bucket sub-problems is to use partially overlapping buckets: this could potentially increase the possibility of related variables to be included together in the kernel set.

We implemented this feature as an alternative bucket construction criterion. This feature works well with certain combinations of kernel construction criterion and variable sorting criterion.

5.4 Ejection of Variables from the Kernel

Another possible improvement for the efficiency of the Kernel Search, is the *kernel eject* method, which allows to remove variables from the kernel. This technique is particularly useful for more complex instances, where every bucket sub-problem normally takes a significant amount of time to be solved.

To implement this functionality, at each iteration of the Kernel Search we keep a counter h for the number of solutions found. Each variable that isn't in the initial kernel is also associated with a counter $k_v, v \in \textit{variables} \setminus \textit{variables in the initial kernel}$ that is incremented when the variable has a non-zero value in the solution of a bucket sub-problem. At the end of each iteration, for each variable v, if $(h - k_v) - k_v <= \textit{threshold}$, the variable is removed from the kernel.

The values of h and k_v are reset at each iteration of the Kernel Search.

The *eject threshold* is a configuration parameter: a low threshold will remove more variables from the kernel, increasing efficiency at the cost of the quality of the solution found. A high threshold instead has a more subtle effect.

When the variables are removed from the kernel they are not deleted from the model, but they are returned to their original bucket.

Experiments on the instances demonstrated that this method significantly cuts the solving time of the bucket sub-problems. The quality of the solution found is worsened, but on the flip side the more iterations and buckets can be solved in the same time limit.

5.5 Repetition Counter

An observation we could make is that when solving buckets, after a certain number of iterations the algorithm repeatedly finds new solutions with the same objective value, and sometimes it even struggles to find new solutions at all. In other words, the algorithm gets stuck in *local optima*, from which it's hard to escape.

Our hypothesis is that there are two causes for this:

1. The GUROBI solver cannot find a (better) solution for the sub-problem, either due to the infeasibility of the problem or because of the time limit set for solving each bucket.
2. The Kernel Search algorithm, during its improvement phase, only accepts a new solution (thus updating the kernel) if it improves upon, or is at least equal, to the incumbent one. This is done by adding a cutoff constraint, as explained in 2.3.

In order to mitigate this problem we introduced a *repetition counter* that, during the improvement phase of Kernel Search, removes the cutoff constraint for k buckets when the same solution (or no solution) is found for h times.

The idea is that this method allows to select if the focus should be on *diversification* or *intensification*, by appropriately setting parameters h and k.

A low value for h and a high one for k allow for diversification, by adding to the kernel variables that otherwise may never be selected, and could allow escaping the local optimum.

On the opposite, a high value for h and a low one for k switch to focus on intensification, by giving priority to finding better solutions.

After experimenting with different values for h and k, we found that keeping them more or less equal allowed for a reasonable balance between diversification and intensification. In particular we found that $h = 3$ and $k = 3$ worked quite well with the test instances we selected.

5.5.1 Reset counter on new optimum

A small enhancement we have applied is to reset the counter to its initial status whenever the Kernel Search finds a new solution that is better than any other found before.

This is significant because the default repetition counter method completely ignores the value of the solutions found during the k cycles.

5.6 Item Dominance

In [1] an improvement for the MKP model is suggested: given two items j and k, if $w_j \leq w_k$ and $p_j \geq p_k$, then it is said that j *dominates* k. This means that when an item is excluded from the solution, all items dominated by it can also be excluded.

The simplest way to implement this method is to preliminarily sort all items according to non-increasing weight, breaking ties by non-decreasing profit. For each item k, items $j := k+1, \ldots, n$ are parsed, and if $p_j \geq p_k$ then the pair (j, k) is added to a dominance list D. Then, the following constraints are added to the model:

$$\sum_{i=1}^{m} x_{ij} \geq \sum_{i=1}^{m} x_{ik} \qquad (j, k) \in D \tag{5.1}$$

To efficiently implement this technique,the dominance list is only built once at startup and the constraints (5.1) are applied when solving the relaxation, the kernel problem and the bucket sub-problems.

The testing proved that this method increases performance of the Kernel Search with very little overhead costs.

5.7 Instance Reduction

Another improvement for the MKP model describe in [1] is *instance reduction*.

Let I be any subset of knapsacks and let J be the set of all items of the instance that can be packed in a knapsack of I:

$$J \coloneqq \{j : w_j \leq \max_{i \in I}\{c_i\}, 1 \leq j \leq n\} \tag{5.2}$$

If there exists a feasible packing of the items of J into the knapsacks of I, then such packing can be fixed, and sets I and J can be removed from the instance.

The most efficient way to implement this property is to start by sorting the knapsacks by non-decreasing capacity, add the first knapsack to set I, and iteratively add the next (smaller) knapsack to I. At each iteration, we add the appropriate items to J, and check if $\sum_{j \in J} w_j \leq \sum_{i \in I} c_i$. This allows to avoid running the more expensive bin packing algorithm when it's guaranteed that there is no feasible solution.

To solve the bin packing problem, in [1] an exact method is suggested. However, the execution time for such an algorithm is unacceptable for this project. We decided anyway to try solving the bin packing problem using a modified version of the *First-Fit-Decreasing (FFD)* heuristic.

The basic version of FFD works as follows: Given n items with a weight and a fixed capacity for each bin:

1. Order the items from largest to smallest;
2. Open a new empty bin;
3. For each item, from largest to smallest, find the first bin into which the item fits, if any. If such a bin is found, put the item in it. Otherwise, open a new empty bin and put the item in it.

In this algorithm the bins correspond to the knapsacks of I, and the items are the ones contained in J. Also, the number of bins is pre-determined by the size of I, and each bin has a different capacity (corresponding to the capacity of the knapsack).

We implemented this algorithm in the code as an additional step to be run before solving the kernel problem and the bucket sub-problems. The testing however revealed that the algorithm couldn't find any valid bin packing for any test instance. After some consideration, we supposed that even if we were to swap FFD with another heuristic algorithm, it would be unlikely that the bin packing found (if any) would impact the efficiency of the Kernel Search by much. Because of this reason, we decided to discard this technique.

5.8 Single Knapsack Heuristic

As an experiment, we tried to implement a heuristic algorithm that, for each knapsack, solves a *0-1 knapsack problem*, which finds the optimum items to be included in that knapsack.

Given n items with a weight w_j and a profit p_j, and a knapsack with capacity c, the *0-1 Knapsack Problem* finds the subset of items that maximizes the overall profit while not overflowing the capacity of the knapsack.

$$\max z = \sum_{j=1}^{n} p_j x_j \tag{5.3}$$

$$\sum_{j=1}^{n} w_j x_j \leq c \tag{5.4}$$

$$x_j \in \{0, 1\} \qquad j = 1, \dots, n \tag{5.5}$$

It's worth noting that the *0-1 Knapsack Problem* is equal to an MKP where $m = 1$;

The detailed algorithm begins with defining $J :=$ items of the MKP and $X := \emptyset$ as the (initially empty) solution found by the heuristic. Then, for each knapsack $i = 1, \dots, m$ the following steps are repeated:

1. Solve the *0-1 Knapsack Problem* that uses i as the knapsack and the items in J;
2. Remove the items that were inserted into the knapsack from J;
3. Add the solution found (associated to the knapsack) to X;

Since every item can at most be inserted into one knapsack, and the capacity of each knapsack is respected, X is guaranteed to be a feasible solution for the MKP.

After preliminarily testing the heuristic by itself (outside the Kernel Search framework) we found that the results were excellent, and required very little computation time.

5.8.1 Integration with the Kernel Search

To integrate the heuristic with the Kernel Search, our idea was to use the heuristic to find a high quality starting solution, which the Kernel Search tries to improve upon.

To achieve this, we removed the solving of the LP relaxation, and instead used the solution found by the heuristic to build the kernel set and the buckets. The parameters of the Kernel Search and the methods added until now did not require any change to work correctly.

It is possible that the Kernel Search does not find a solution that improves upon the one found by the heuristic: this is still acceptable, because the heuristic finds integer and feasible solutions.

6. Testing

In this chapter we will report the results of the tests executed on the instances introduced in 4.1.2, to verify the performance of the tuned Kernel Search algorithm.

6.1 Default Bucket Construction Criterion

The objective of the first round of testing was to find the most efficient combination of kernel construction criterion and variable sorting criterion, using the default bucket construction criterion (non-overlapping buckets). Certain variable sorting criteria work really well with certain kernel construction criteria, but give bad results with others.

We discovered that the two sorting criteria that worked best were to sort by *value, profit and weight, breaking ties using the reduced cost*, and the random sorting. Random sorting however, due to its nature, produces inconsistent results.

The tables from 6.1 to 6.4 contain the test results of all the kernel construction criteria introduced in 5.2 combined with the sorting by value, profit, weight and reduced cost.

The complete test results, including the ones with other sorting criteria, can be found at `https://github.com/Golino98/KernelSearchGolinoCottiBeatrice/tree/main/log/DefaultBucket`.

In general, the performance of the criteria is comparable, with the integer kernel being slightly more reliable.

6.2 Overlapping buckets

In this section we report the results of the tests executed with the overlapping buckets. Just like in the section before, we executed the tests for each possible combination of kernel builder criterion and variable sorting criterion.

Tables from 6.5 to 6.8 represent the results of the tests run with the various kernel builders, and the variables sorted by *value, profit and weight, breaking ties using the reduced cost*. All the other tests results can be found at `https://github.com/Golino98/KernelSearchGolino`

Directory	Instance	OPT	Time Elapsed	Time limit Reached
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15431	10.7006185	false
	probT1_0U_R50_T002_M020_N0060_seed01	26025	18.140957399	false
	probT1_1W_R50_T002_M010_N0040_seed09	12724	38.408752099	false
	probT1_1W_R50_T002_M010_N0060_seed10	19610	120.743802999	true
	probT1_1W_R50_T002_M020_N0020_seed10	3429	4.303161301	false
	probT1_1W_R50_T002_M020_N0040_seed10	12091	11.542264499	false
FK_1	random10_100_2_1000_1_10	29767	47.463179701	false
	random10_60_1_1000_1_12	23064	118.2782449	true
	random12_48_3_1000_1_16	11845	14.1033361	false
	random15_45_1_1000_1_13	14927	12.9597833	false
	random15_75_3_1000_1_16	18298	118.4746604	true
	random30_60_4_1000_1_14	11017	33.0105262	false
FK_2	random20_120_1_1000_1_12	47752	118.0889282	true
	random20_200_2_1000_1_14	53465	120.160886601	true
	random24_96_3_1000_1_18	23868	118.2763374	true
	random30_150_2_1000_1_17	44255	118.296452299	true
	random30_90_4_1000_1_20	26023	62.8627581	false
	random60_120_4_1000_1_19	20463	118.0074992	true
FK_3\notSolved	random30_180_2_1000_1_2	47791	120.476056301	true
	random36_144_2_1000_1_20	42519	118.062551601	true
	random45_135_3_1000_1_12	37583	118.0600674	true
	random45_135_4_1000_1_14	33624	118.068143999	true
	random45_225_2_1000_1_20	67704	118.159034901	true
FK_3\Solved	random30_180_1_1000_1_20	75311	120.5786256	true
	random30_300_2_1000_1_4	84842	120.3795314	true
	random36_144_3_1000_1_17	39150	121.053216999	true
	random45_135_4_1000_1_16	36976	118.2337826	true
	random45_225_3_1000_1_15	56015	120.5805333	true
	random90_180_4_1000_1_19	30047	118.9785758	true
FK_4\notSolved	random50_300_2_1000_1_5	80067	121.506050199	true
	random60_240_1_1000_1_17	95078	119.386242201	true
	random60_240_2_1000_1_15	66025	120.5330838	true
	random60_240_3_1000_1_18	59371	121.443567401	true
	random75_225_4_1000_1_20	58309	119.0543748	true
FK_4\Solved	random150_300_2_1000_1_12	54379	121.2856578	true
	random50_300_1_1000_1_16	119257	121.140542999	true
	random50_500_2_1000_1_14	134766	122.3455152	true
	random60_240_3_1000_1_14	60875	121.6984316	true
	random75_225_4_1000_1_17	60417	118.714197001	true
	random75_375_2_1000_1_5	101186	122.4572583	true

Table 6.1: Results of Kernel builder positive with variables sorted by profit, weight and RC

Directory	Instance	OPT	Time Elapsed	Time limit Reached
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15440	118.6790482	true
	probT1_0U_R50_T002_M020_N0060_seed01	25636	19.5386964	false
	probT1_1W_R50_T002_M010_N0040_seed09	12699	9.7501646	false
	probT1_1W_R50_T002_M010_N0060_seed10	19610	118.1473723	true
	probT1_1W_R50_T002_M020_N0020_seed10	3429	4.4997488	false
	probT1_1W_R50_T002_M020_N0040_seed10	12233	13.5511261	false
FK_1	random10_100_2_1000_1_10	29714	121.0209832	true
	random10_60_1_1000_1_12	23064	58.3368111	false
	random12_48_3_1000_1_16	11837	38.1227443	false
	random15_45_1_1000_1_13	14726	10.0408741	false
	random15_75_3_1000_1_16	18296	120.3734505	true
	random30_60_4_1000_1_14	11017	37.1950626	false
FK_2	random20_120_1_1000_1_12	47785	118.9296709	true
	random20_200_2_1000_1_14	53329	120.8207453	true
	random24_96_3_1000_1_18	23848	120.791078	true
	random30_150_2_1000_1_17	44176	118.2500227	true
	random30_90_4_1000_1_20	26021	102.2652646	false
	random60_120_4_1000_1_19	20463	118.2168326	true
FK_3\notSolved	random30_180_2_1000_1_2	47636	120.5323994	true
	random36_144_2_1000_1_20	42594	118.1922323	true
	random45_135_3_1000_1_12	37558	118.2217203	true
	random45_135_4_1000_1_14	33709	120.5539263	true
	random45_225_2_1000_1_20	67597	118.0189444	true
FK_3\Solved	random30_180_1_1000_1_20	75293	121.1674745	true
	random30_300_2_1000_1_4	84727	121.3198455	true
	random36_144_3_1000_1_17	39128	118.8654796	true
	random45_135_4_1000_1_16	36959	121.1141717	true
	random45_225_3_1000_1_15	55862	120.4682651	true
	random90_180_4_1000_1_19	30047	118.5293064	true
FK_4\notSolved	random50_300_2_1000_1_5	79903	121.5573043	true
	random60_240_1_1000_1_17	95156	118.046458	true
	random60_240_2_1000_1_15	65765	121.6410656	true
	random60_240_3_1000_1_18	59294	120.3939231	true
	random75_225_4_1000_1_20	58027	121.1534583	true
FK_4\Solved	random150_300_2_1000_1_12	55101	118.5865302	true
	random50_300_1_1000_1_16	119327	121.077851	true
	random50_500_2_1000_1_14	134389	119.2635756	true
	random60_240_3_1000_1_14	60746	121.4089895	true
	random75_225_4_1000_1_17	56581	122.1691686	true
	random75_375_2_1000_1_5	100595	122.6023048	true

Table 6.2: Results of Kernel builder percentage with variables sorted by value, profit, weight and RC

Directory	Instance	OPT	Time Elapsed	Time limit Reached
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15431	14.3882406	false
	probT1_0U_R50_T002_M020_N0060_seed01	25873	17.6503805	false
	probT1_1W_R50_T002_M010_N0040_seed09	12626	23.8001573	false
	probT1_1W_R50_T002_M010_N0060_seed10	19601	69.6079519	false
	probT1_1W_R50_T002_M020_N0020_seed10	3429	4.1287237	false
	probT1_1W_R50_T002_M020_N0040_seed10	11951	10.3159794	false
FK_1	random10_100_2_1000_1_10	29697	118.128767	true
	random10_60_1_1000_1_12	22907	18.3393749	false
	random12_48_3_1000_1_16	11813	52.8908991	false
	random15_45_1_1000_1_13	14457	8.6747187	false
	random15_75_3_1000_1_16	18298	118.0714965	true
	random30_60_4_1000_1_14	11017	29.0765736	false
FK_2	random20_120_1_1000_1_12	47785	118.040946	true
	random20_200_2_1000_1_14	53481	118.0136104	true
	random24_96_3_1000_1_18	23848	118.044707	true
	random30_150_2_1000_1_17	44240	118.0516259	true
	random30_90_4_1000_1_20	25977	55.9535786	false
	random60_120_4_1000_1_19	20463	118.0305284	true
FK_3\notSolved	random30_180_2_1000_1_2	47785	118.5893507	true
	random36_144_2_1000_1_20	42669	118.173696499	true
	random45_135_3_1000_1_12	37575	118.2469295	true
	random45_135_4_1000_1_14	33665	118.1473182	true
	random45_225_2_1000_1_20	67622	118.2277412	true
FK_3\Solved	random30_180_1_1000_1_20	75357	120.316433199	true
	random30_300_2_1000_1_4	84845	119.340264801	true
	random36_144_3_1000_1_17	39159	120.5446257	true
	random45_135_4_1000_1_16	36953	118.141555501	true
	random45_225_3_1000_1_15	55974	120.981253	true
	random90_180_4_1000_1_19	30047	118.4051206	true
FK_4\notSolved	random50_300_2_1000_1_5	80147	119.3470366	true
	random60_240_1_1000_1_17	95186	118.8712886	true
	random60_240_2_1000_1_15	65879	120.4329632	true
	random60_240_3_1000_1_18	59406	120.8996258	true
	random75_225_4_1000_1_20	58306	118.4224734	true
FK_4\Solved	random150_300_2_1000_1_12	54551	121.6945088	true
	random50_300_1_1000_1_16	119577	120.0297411	true
	random50_500_2_1000_1_14	134218	119.8665574	true
	random60_240_3_1000_1_14	60900	121.5346966	true
	random75_225_4_1000_1_17	60409	118.6410517	true
	random75_375_2_1000_1_5	100643	118.6486458	true

Table 6.3: Results of Kernel builder with integer variables sorted by profit, weight and absolute RC

Directory	Instance	OPT	Time Elapsed	Time limit Reached
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15192	7.4258463	false
	probT1_0U_R50_T002_M020_N0060_seed01	25580	17.4606988	false
	probT1_1W_R50_T002_M010_N0040_seed09	12632	10.6713191	false
	probT1_1W_R50_T002_M010_N0060_seed10	19576	118.7736707	true
	probT1_1W_R50_T002_M020_N0020_seed10	3429	5.4714043	false
	probT1_1W_R50_T002_M020_N0040_seed10	12167	11.2754993	false
FK_1	random10_100_2_1000_1_10	29782	64.981783	false
	random10_60_1_1000_1_12	23021	24.5501013	false
	random12_48_3_1000_1_16	11835	21.5266961	false
	random15_45_1_1000_1_13	14457	8.4832163	false
	random15_75_3_1000_1_16	18281	119.3678719	true
	random30_60_4_1000_1_14	11017	32.2837986	false
FK_2	random20_120_1_1000_1_12	47813	71.8129364	false
	random20_200_2_1000_1_14	53478	121.100354	true
	random24_96_3_1000_1_18	23860	120.4599256	true
	random30_150_2_1000_1_17	44235	118.0590868	true
	random30_90_4_1000_1_20	25981	55.6203358	false
	random60_120_4_1000_1_19	20463	118.0701523	true
FK_3\notSolved	random30_180_2_1000_1_2	47780	120.9997216	true
	random36_144_2_1000_1_20	42672	118.1081382	true
	random45_135_3_1000_1_12	37531	118.2918107	true
	random45_135_4_1000_1_14	33687	118.001285	true
	random45_225_2_1000_1_20	67661	119.0418638	true
FK_3\Solved	random30_180_1_1000_1_20	75487	120.4264408	true
	random30_300_2_1000_1_4	84882	120.5467055	true
	random36_144_3_1000_1_17	39181	120.0839264	true
	random45_135_4_1000_1_16	36937	118.0784009	true
	random45_225_3_1000_1_15	56013	120.9657798	true
	random90_180_4_1000_1_19	30047	118.5214583	true
FK_4\notSolved	random50_300_2_1000_1_5	80088	121.3204107	true
	random60_240_1_1000_1_17	95273	118.5298356	true
	random60_240_2_1000_1_15	66063	121.1171312	true
	random60_240_3_1000_1_18	59433	119.7835869	true
	random75_225_4_1000_1_20	58307	118.0072072	true
FK_4\Solved	random150_300_2_1000_1_12	54490	120.7769111	true
	random50_300_1_1000_1_16	119365	121.85657	true
	random50_500_2_1000_1_14	134744	121.8521347	true
	random60_240_3_1000_1_14	60774	121.3412287	true
	random75_225_4_1000_1_17	60427	118.4249445	true
	random75_375_2_1000_1_5	100607	124.3004088	true

Table 6.4: Results of Kernel builder with a threshold limit. Variables are sorted by value, profit, weight and RC.

`CottiBeatrice/tree/main/log/BucketOverlap`.

The performance of the overlapping buckets on average slightly better (but still very much comparable) to the non-overlapping ones.

Directory	Instance	OPT	Time Elapsed	Time Limit Reached
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15431	10.5499199	false
	probT1_0U_R50_T002_M020_N0060_seed01	26025	17.8285911	false
	probT1_1W_R50_T002_M010_N0040_seed09	12724	38.6574932	false
	probT1_1W_R50_T002_M010_N0060_seed10	19589	120.6389994	true
	probT1_1W_R50_T002_M020_N0020_seed10	3429	4.227292	false
	probT1_1W_R50_T002_M020_N0040_seed10	12091	11.4228777	false
FK_1	random10_100_2_1000_1_10	29767	47.8715247	false
	random10_60_1_1000_1_12	23064	118.1510923	true
	random12_48_3_1000_1_16	11845	14.0335003	false
	random15_45_1_1000_1_13	14927	12.9012964	false
	random15_75_3_1000_1_16	18298	118.0100516	true
	random30_60_4_1000_1_14	11017	32.1676649	false
FK_2	random20_120_1_1000_1_12	47752	120.7777181	true
	random20_200_2_1000_1_14	53465	119.0939042	true
	random24_96_3_1000_1_18	23868	118.0726257	true
	random30_150_2_1000_1_17	44255	118.0677943	true
	random30_90_4_1000_1_20	26023	62.2006093	false
	random60_120_4_1000_1_19	20463	118.1230571	true
FK_3\notSolved	random30_180_2_1000_1_2	47799	119.6106118	true
	random36_144_2_1000_1_20	42519	118.1420776	true
	random45_135_3_1000_1_12	37583	118.1477843	true
	random45_135_4_1000_1_14	33624	118.1360589	true
	random45_225_2_1000_1_20	67704	118.263307	true
FK_3\Solved	random30_180_1_1000_1_20	75311	119.2969998	true
	random30_300_2_1000_1_4	84852	120.6146807	true
	random36_144_3_1000_1_17	39150	119.024031	true
	random45_135_4_1000_1_16	36976	118.1475608	true
	random45_225_3_1000_1_15	56052	121.4289919	true
	random90_180_4_1000_1_19	30047	119.1823311	true
FK_4\notSolved	random50_300_2_1000_1_5	80096	120.6236929	true
	random60_240_1_1000_1_17	95078	118.7310264	true
	random60_240_2_1000_1_15	66025	120.924096	true
	random60_240_3_1000_1_18	59361	121.5133865	true
	random75_225_4_1000_1_20	58309	118.0206647	true
FK_4\Solved	random150_300_2_1000_1_12	54379	119.2619161	true
	random50_300_1_1000_1_16	119243	120.5774003	true
	random50_500_2_1000_1_14	134766	121.9378357	true
	random60_240_3_1000_1_14	60875	121.0021095	true
	random75_225_4_1000_1_17	60417	118.4098234	true
	random75_375_2_1000_1_5	101186	122.5474228	true

Table 6.5: Results of Kernel builder positive with variables sorted by profit, weight and RC. Buckets can overlap.

Directory	Instance	OPT	Time Elapsed	Time Limit Reached
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15440	118.2327952	true
	probT1_0U_R50_T002_M020_N0060_seed01	25636	19.2754894	false
	probT1_1W_R50_T002_M010_N0040_seed09	12699	9.6102967	false
	probT1_1W_R50_T002_M010_N0060_seed10	19610	119.6537668	true
	probT1_1W_R50_T002_M020_N0020_seed10	3429	4.4839481	false
	probT1_1W_R50_T002_M020_N0040_seed10	12233	13.6189164	false
FK_1	random10_100_2_1000_1_10	29714	120.9370562	true
	random10_60_1_1000_1_12	23064	59.4954217	false
	random12_48_3_1000_1_16	11837	40.4503144	false
	random15_45_1_1000_1_13	14726	9.739517	false
	random15_75_3_1000_1_16	18296	120.376264	true
	random30_60_4_1000_1_14	11017	37.502175	false
FK_2	random20_120_1_1000_1_12	47785	118.6698652	true
	random20_200_2_1000_1_14	53329	120.5996822	true
	random24_96_3_1000_1_18	23848	118.4356063	true
	random30_150_2_1000_1_17	44176	118.1388416	true
	random30_90_4_1000_1_20	26021	102.4707351	false
	random60_120_4_1000_1_19	20463	118.2593343	true
FK_3\notSolved	random30_180_2_1000_1_2	47650	121.0552006	true
	random36_144_2_1000_1_20	42594	118.6566244	true
	random45_135_3_1000_1_12	37558	118.248681	true
	random45_135_4_1000_1_14	33709	118.378746	true
	random45_225_2_1000_1_20	67631	119.2291363	true
FK_3\Solved	random30_180_1_1000_1_20	75293	120.9974902	true
	random30_300_2_1000_1_4	84727	121.1806452	true
	random36_144_3_1000_1_17	39133	119.5650327	true
	random45_135_4_1000_1_16	36964	120.774835	true
	random45_225_3_1000_1_15	55866	120.4736737	true
	random90_180_4_1000_1_19	30047	118.4590028	true
FK_4\notSolved	random50_300_2_1000_1_5	79848	121.1269227	true
	random60_240_1_1000_1_17	95156	121.5917362	true
	random60_240_2_1000_1_15	65765	121.4508213	true
	random60_240_3_1000_1_18	59315	119.3413821	true
	random75_225_4_1000_1_20	58027	121.4900267	true
FK_4\Solved	random150_300_2_1000_1_12	50643	119.7120855	true
	random50_300_1_1000_1_16	119513	121.102601	true
	random50_500_2_1000_1_14	134389	120.4341443	true
	random60_240_3_1000_1_14	60763	118.6870838	true
	random75_225_4_1000_1_17	55073	121.5284846	true
	random75_375_2_1000_1_5	100595	122.6528904	true

Table 6.6: Results of Kernel builder percentage with variables sorted by value, profit, weight and RC. Buckets can overlap.

Directory	Instance	OPT	Time Elapsed	Is the optimum
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15431	14.0181948	false
	probT1_0U_R50_T002_M020_N0060_seed01	25873	17.6749925	false
	probT1_1W_R50_T002_M010_N0040_seed09	12626	23.8664822	false
	probT1_1W_R50_T002_M010_N0060_seed10	19601	72.3460785	false
	probT1_1W_R50_T002_M020_N0020_seed10	3429	4.0769064	false
	probT1_1W_R50_T002_M020_N0040_seed10	11951	10.4874044	false
FK_1	random10_100_2_1000_1_10	29697	118.0022093	true
	random10_60_1_1000_1_12	22907	18.4083546	false
	random12_48_3_1000_1_16	11813	53.0097031	false
	random15_45_1_1000_1_13	14457	8.5033898	false
	random15_75_3_1000_1_16	18298	118.1824136	true
	random30_60_4_1000_1_14	11017	29.2174872	false
FK_2	random20_120_1_1000_1_12	47785	118.8841301	true
	random20_200_2_1000_1_14	53481	118.0334493	true
	random24_96_3_1000_1_18	23848	118.0280233	true
	random30_150_2_1000_1_17	44240	118.1493584	true
	random30_90_4_1000_1_20	25977	55.7077807	false
	random60_120_4_1000_1_19	20463	118.0352109	true
FK_3\notSolved	random30_180_2_1000_1_2	47785	118.1238304	true
	random36_144_2_1000_1_20	42669	118.0920157	true
	random45_135_3_1000_1_12	37575	118.1369549	true
	random45_135_4_1000_1_14	33665	118.1448202	true
	random45_225_2_1000_1_20	67622	118.1296609	true
FK_3\Solved	random30_180_1_1000_1_20	75357	120.6022605	true
	random30_300_2_1000_1_4	84871	120.4607194	true
	random36_144_3_1000_1_17	39159	118.8372864	true
	random45_135_4_1000_1_16	36953	118.1114857	true
	random45_225_3_1000_1_15	55974	121.145959	true
	random90_180_4_1000_1_19	30047	118.4031226	true
FK_4\notSolved	random50_300_2_1000_1_5	80135	118.2987666	true
	random60_240_1_1000_1_17	95186	118.2756981	true
	random60_240_2_1000_1_15	65874	118.4979199	true
	random60_240_3_1000_1_18	59406	120.9757263	true
	random75_225_4_1000_1_20	58307	118.2047939	true
FK_4\Solved	random150_300_2_1000_1_12	54551	118.2603937	true
	random50_300_1_1000_1_16	119577	120.8787226	true
	random50_500_2_1000_1_14	134942	121.6452096	true
	random60_240_3_1000_1_14	60900	119.4260313	true
	random75_225_4_1000_1_17	60411	118.1878376	true
	random75_375_2_1000_1_5	100975	122.7270384	true

Table 6.7: Results of Kernel builder with integer variables sorted by profit, weight and absolute RC. Buckets can overlap.

Directory	Instance	OPT	Time Elapsed	Time Limit Reached
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15121	7.1886766	false
	probT1_0U_R50_T002_M020_N0060_seed01	25987	12.4775836	false
	probT1_1W_R50_T002_M010_N0040_seed09	12699	15.787580401	false
	probT1_1W_R50_T002_M010_N0060_seed10	19551	21.8028844	false
	probT1_1W_R50_T002_M020_N0020_seed10	3015	3.243724399	false
	probT1_1W_R50_T002_M020_N0040_seed10	12195	7.8216764	false
FK_1	random10_100_2_1000_1_10	29743	118.5247869	true
	random10_60_1_1000_1_12	23064	33.6302586	false
	random12_48_3_1000_1_16	11830	15.9524401	false
	random15_45_1_1000_1_13	14926	6.6863515	false
	random15_75_3_1000_1_16	18291	47.7703098	false
	random30_60_4_1000_1_14	11017	18.9787416	false
FK_2	random20_120_1_1000_1_12	47679	54.5706415	false
	random20_200_2_1000_1_14	53455	120.9019194	true
	random24_96_3_1000_1_18	23829	118.0846664	true
	random30_150_2_1000_1_17	44177	118.0273574	true
	random30_90_4_1000_1_20	25968	32.4749387	false
	random60_120_4_1000_1_19	20463	118.0244715	true
FK_3\notSolved	random30_180_2_1000_1_2	47777	118.0200626	true
	random36_144_2_1000_1_20	42442	90.8936418	false
	random45_135_3_1000_1_12	37582	118.0143393	true
	random45_135_4_1000_1_14	33708	106.947772	false
	random45_225_2_1000_1_20	67808	118.4547918	true
FK_3\Solved	random30_180_1_1000_1_20	75379	120.9968081	true
	random30_300_2_1000_1_4	84886	120.2451406	true
	random36_144_3_1000_1_17	39159	118.672092	true
	random45_135_4_1000_1_16	36918	101.4291408	false
	random45_225_3_1000_1_15	55935	121.1715867	true
	random90_180_4_1000_1_19	30047	118.2124253	true
FK_4\notSolved	random50_300_2_1000_1_5	80171	120.2989442	true
	random60_240_1_1000_1_17	95353	121.160608899	true
	random60_240_2_1000_1_15	66076	118.1332428	true
	random60_240_3_1000_1_18	59329	120.592577701	true
	random75_225_4_1000_1_20	58315	118.0759211	true
FK_4\Solved	random150_300_2_1000_1_12	55315	119.7771642	true
	random50_300_1_1000_1_16	119659	118.2373015	true
	random50_500_2_1000_1_14	135045	121.074897701	true
	random60_240_3_1000_1_14	60961	121.1508002	true
	random75_225_4_1000_1_17	60449	118.2291751	true
	random75_375_2_1000_1_5	101096	121.4945341	true

Table 6.8: Results of Kernel builder with a threshold limit. Variables are sorted by value, profit, weight and RC. Buckets can overlap.

6.3 Specific Improvements

Using the results of the test above, we were able to determine for each kernel construction criterion the most efficient combination of variable sorting criterion and bucket construction criterion (overlapping or non-overlapping buckets).

In the following sections, we report the performance of each improvement explained in chapter 5, using the best possible configuration for each kernel construction criterion.

To simplify the testing, each improvement was tested by itself: this means that a test suite was run for each improvement, where all the other improvements are disabled.

Once again, not all test results are reported: the complete set can be found at `https://github.com/Golino98/KernelSearchGolinoCottiBeatrice/tree/main/log`.

6.3.1 Integer Kernel

The best configuration for this kernel construction criterion is to use overlapping bucket, and to sort variables by value, profit, weight and reduced cost.

6.3.2 Percentage Kernel

For this kernel construction criterion the best configuration uses overlapping buckets, and sorts variables by reduced cost and value. In general, we have noticed that this configuration behaves extremely well for small instances, but quite poorly on bigger ones.

6.3.3 Positive Kernel

We have found three equally good configurations for this kernel construction criterion. The one that gave slightly better results is the one that uses non-overlapping buckets and the random sorter. It is worth noting however that, since variables are sorted randomly, the results may vary with each execution.

6.3.4 Threshold Kernel

The best configuration for the kernel construction that select variables based on a threshold is the one that uses overlapping buckets and sort by value, profit, weight and reduced cost.

Directory	Instance	Kernel Int Var	Kernel Percentage	Kernel Positive	Kernel Threshold
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15431	15431	15431	15431
	probT1_0U_R50_T002_M020_N0060_seed01	25580	26502	26250	25580
	probT1_1W_R50_T002_M010_N0040_seed09	12582	12724	12528	12664
	probT1_1W_R50_T002_M010_N0060_seed10	19584	19651	19594	19559
	probT1_1W_R50_T002_M020_N0020_seed10	3429	3429	3429	3429
	probT1_1W_R50_T002_M020_N0040_seed10	12233	12335	12153	11772
FK_1	random10_100_2_1000_1_10	29703	29749	29708	29782
	random10_60_1_1000_1_12	22950	23064	23002	29029
	random12_48_3_1000_1_16	11818	11817	11736	11810
	random15_45_1_1000_1_13	14811	15081	15083	14811
	random15_75_3_1000_1_16	18282	18295	18265	18284
	random30_60_4_1000_1_14	11017	11017	11017	11017
FK_2	random20_120_1_1000_1_12	47760	47752	47667	47760
	random20_200_2_1000_1_14	53501	53369	53309	53504
	random24_96_3_1000_1_18	23853	23835	23795	23837
	random30_150_2_1000_1_17	44204	44395	44103	44187
	random30_90_4_1000_1_20	25962	25997	25927	25983
	random60_120_4_1000_1_19	20463	20463	20463	19467
FK_3\notSolved	random30_180_2_1000_1_2	47685	47554	47752	47828
	random36_144_2_1000_1_20	42548	42742	42368	42535
	random45_135_3_1000_1_12	37575	37534	37433	37519
	random45_135_4_1000_1_14	33673	33719	33669	33558
	random45_225_2_1000_1_20	67647	66990	67539	67728
FK_3\Solved	random30_180_1_1000_1_20	75405	73557	75351	75255
	random30_300_2_1000_1_4	84974	84935	84734	84889
	random36_144_3_1000_1_17	39153	38727	39133	39180
	random45_135_4_1000_1_16	36844	36958	36919	36867
	random45_225_3_1000_1_15	55869	55399	56022	56002
	random90_180_4_1000_1_19	30047	30047	30047	30047
FK_4\notSolved	random50_300_2_1000_1_5	80146	78336	79720	79983
	random60_240_1_1000_1_17	95345	94488	94982	95339
	random60_240_2_1000_1_15	65947	65852	65941	66037
	random60_240_3_1000_1_18	59302	57339	59346	59398
	random75_225_4_1000_1_20	58137	58016	58272	58308
FK_4\Solved	random150_300_2_1000_1_12	55677	33561	55703	54404
	random50_300_1_1000_1_16	119505	118806	119195	119557
	random50_500_2_1000_1_14	134971	133863	134947	135017
	random60_240_3_1000_1_14	60985	60400	60907	60849
	random75_225_4_1000_1_17	60436	56633	60386	60430
	random75_375_2_1000_1_5	101040	99489	100843	100706

Table 6.9: Comparison of the eject improvement between different Kernel configurations.

Directory	Instance	Kernel Int Var	Kernel Percentage	Kernel Positive	Kernel Threshold
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15431	15478	15431	15440
	probT1_0U_R50_T002_M020_N0060_seed01	25846	26502	26236	25829
	probT1_1W_R50_T002_M010_N0040_seed09	12699	12724	12675	12657
	probT1_1W_R50_T002_M010_N0060_seed10	19650	19651	19650	19650
	probT1_1W_R50_T002_M020_N0020_seed10	3429	3429	3429	3429
	probT1_1W_R50_T002_M020_N0040_seed10	12195	12339	12235	12237
FK_1	random10_100_2_1000_1_10	29764	29749	29775	29767
	random10_60_1_1000_1_12	23029	23064	23064	23034
	random12_48_3_1000_1_16	11837	11845	11812	11840
	random15_45_1_1000_1_13	14832	15160	15081	15115
	random15_75_3_1000_1_16	18312	18302	18288	18295
	random30_60_4_1000_1_14	11017	11017	11017	11017
FK_2	random20_120_1_1000_1_12	47760	47785	47752	47760
	random20_200_2_1000_1_14	53485	53359	53382	53478
	random24_96_3_1000_1_18	23867	23841	23849	23871
	random30_150_2_1000_1_17	44195	44395	44301	44235
	random30_90_4_1000_1_20	26002	26022	25989	25987
	random60_120_4_1000_1_19	19923	20463	20463	20463
FK_3\notSolved	random30_180_2_1000_1_2	47833	47554	47743	47746
	random36_144_2_1000_1_20	42583	42772	42677	42627
	random45_135_3_1000_1_12	37601	37507	37531	37576
	random45_135_4_1000_1_14	33705	33685	33696	33688
	random45_225_2_1000_1_20	67730	66977	67700	67747
FK_3\Solved	random30_180_1_1000_1_20	75460	73557	75245	75490
	random30_300_2_1000_1_4	84943	84825	84776	84882
	random36_144_3_1000_1_17	39195	38991	39155	39181
	random45_135_4_1000_1_16	36941	36963	36948	36946
	random45_225_3_1000_1_15	55935	55428	55963	55982
	random90_180_4_1000_1_19	30047	30047	30047	30047
FK_4\notSolved	random50_300_2_1000_1_5	80187	78336	79711	80088
	random60_240_1_1000_1_17	95352	95060	95137	95273
	random60_240_2_1000_1_15	66057	65871	65852	66068
	random60_240_3_1000_1_18	59351	59385	59372	53433
	random75_225_4_1000_1_20	58315	58016	58288	58312
FK_4\Solved	random150_300_2_1000_1_12	55681	38290	55689	54300
	random50_300_1_1000_1_16	119476	118806	119197	119539
	random50_500_2_1000_1_14	134932	133902	134866	135012
	random60_240_3_1000_1_14	60963	60477	60938	60818
	random75_225_4_1000_1_17	60449	56633	60426	60432
	random75_375_2_1000_1_5	101242	99489	100735	100873

Table 6.10: Comparison of the repetition counter improvement between different Kernel configurations.

Directory	Instance	Kernel Int Var	Kernel Percentage	Kernel Positive	Kernel Threshold
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15431	15431	15436	15431
	probT1_0U_R50_T002_M020_N0060_seed01	26415	26502	26502	26280
	probT1_1W_R50_T002_M010_N0040_seed09	12538	12724	12724	12528
	probT1_1W_R50_T002_M010_N0060_seed10	19650	19651	19650	19519
	probT1_1W_R50_T002_M020_N0020_seed10	3429	3429	3429	3429
	probT1_1W_R50_T002_M020_N0040_seed10	11447	12091	12124	11776
FK_1	random10_100_2_1000_1_10	29767	29723	29726	29727
	random10_60_1_1000_1_12	22992	23064	23064	23064
	random12_48_3_1000_1_16	11831	11847	11729	11821
	random15_45_1_1000_1_13	15081	15053	14753	14680
	random15_75_3_1000_1_16	18298	18258	18280	18255
	random30_60_4_1000_1_14	11017	11017	11017	11017
FK_2	random20_120_1_1000_1_12	47724	47696	47689	47667
	random20_200_2_1000_1_14	53502	53495	53286	53526
	random24_96_3_1000_1_18	23839	23787	23850	23867
	random30_150_2_1000_1_17	44198	44196	44255	44154
	random30_90_4_1000_1_20	26016	25998	25959	25993
	random60_120_4_1000_1_19	20463	20463	20463	20463
FK_3\notSolved	random30_180_2_1000_1_2	47607	47616	47566	47738
	random36_144_2_1000_1_20	42457	42714	42515	42411
	random45_135_3_1000_1_12	37511	37417	37496	37532
	random45_135_4_1000_1_14	33704	33732	33690	33603
	random45_225_2_1000_1_20	67634	67366	67416	67506
FK_3\Solved	random30_180_1_1000_1_20	75377	75179	75136	75334
	random30_300_2_1000_1_4	84907	84551	84638	84758
	random36_144_3_1000_1_17	39179	38702	39157	39184
	random45_135_4_1000_1_16	36919	36942	36927	36953
	random45_225_3_1000_1_15	55982	54930	55933	55922
	random90_180_4_1000_1_19	30047	30044	30047	30047
FK_4\notSolved	random50_300_2_1000_1_5	80154	79904	79512	79954
	random60_240_1_1000_1_17	95166	94395	94832	95148
	random60_240_2_1000_1_15	65935	64968	65754	65772
	random60_240_3_1000_1_18	59513	54387	59389	59256
	random75_225_4_1000_1_20	58308	58028	58272	58311
FK_4\Solved	random150_300_2_1000_1_12	36076	37737	53976	35618
	random50_300_1_1000_1_16	119515	117854	118976	119342
	random50_500_2_1000_1_14	134497	134414	134804	134412
	random60_240_3_1000_1_14	60915	59241	60926	60874
	random75_225_4_1000_1_17	60409	59733	60409	60424
	random75_375_2_1000_1_5	101006	95773	100447	100494

Table 6.11: Comparison of the item dominance improvement between different Kernel configurations.

Directory	Instance	Kernel Int Var	Kernel Percentage	Kernel Positive	Kernel Threshold
SMALL	probT1_0U_R50_T002_M010_N0040_seed05	15478	15534	15478	15534
	probT1_0U_R50_T002_M020_N0060_seed01	26593	26593	26593	26593
	probT1_1W_R50_T002_M010_N0040_seed09	12724	12724	12724	12724
	probT1_1W_R50_T002_M010_N0060_seed10	19651	19650	19651	19650
	probT1_1W_R50_T002_M020_N0020_seed10	3429	3429	3429	3429
	probT1_1W_R50_T002_M020_N0040_seed10	12340	12405	12340	12274
FK_1	random10_100_2_1000_1_10	29738	29741	29762	29738
	random10_60_1_1000_1_12	23064	23064	23064	23064
	random12_48_3_1000_1_16	11844	11845	11816	11845
	random15_45_1_1000_1_13	15160	15160	15115	15160
	random15_75_3_1000_1_16	18292	18296	18285	18296
	random30_60_4_1000_1_14	11017	11017	11017	11017
FK_2	random20_120_1_1000_1_12	47760	47795	47629	47760
	random20_200_2_1000_1_14	53547	53540	53540	53544
	random24_96_3_1000_1_18	23833	23826	23829	23819
	random30_150_2_1000_1_17	44183	44117	44106	44170
	random30_90_4_1000_1_20	25980	25970	25967	25971
	random60_120_4_1000_1_19	20463	20463	20463	20463
FK_3\notSolved	random30_180_2_1000_1_2	47736	47710	47747	47416
	random36_144_2_1000_1_20	42607	42530	42430	42679
	random45_135_3_1000_1_12	37440	37473	37429	37498
	random45_135_4_1000_1_14	33637	33633	33564	33646
	random45_225_2_1000_1_20	67642	67545	67490	67666
FK_3\Solved	random30_180_1_1000_1_20	75304	75306	75281	75379
	random30_300_2_1000_1_4	85044	85044	85044	85044
	random36_144_3_1000_1_17	39158	39145	39097	39159
	random45_135_4_1000_1_16	36895	36904	36772	36872
	random45_225_3_1000_1_15	56119	56060	56050	56086
	random90_180_4_1000_1_19	30047	30047	30047	30047
FK_4\notSolved	random50_300_2_1000_1_5	80228	80228	80228	80228
	random60_240_1_1000_1_17	95312	94961	94872	95312
	random60_240_2_1000_1_15	65717	65855	65654	65843
	random60_240_3_1000_1_18	59561	59520	59520	59520
	random75_225_4_1000_1_20	58231	58247	58231	58231
FK_4\Solved	random150_300_2_1000_1_12	55707	55707	55216	55707
	random50_300_1_1000_1_16	119405	119611	119524	199405
	random50_500_2_1000_1_14	135489	135473	135473	135473
	random60_240_3_1000_1_14	60985	60992	60864	60934
	random75_225_4_1000_1_17	60394	60394	60394	60394
	random75 _375_2_1000_1_5	101242	101242	101242	101242

Table 6.12: Comparison of the heuristic improvement between different Kernel configurations.

7. Conclusions

In this report we have described various improvements that we developed to better tune the Kernel Search algorithm to the MKP.

Some of these improvements, such as Item Dominance and the Single Knapsack Heuristic improve the performance and quality of the algorithm, and thus can be seen as an improvement across the board. Others, such as the Instance reduction and the Repetition Counter usually worsen the results obtained but, depending on how they are configured, they allow to find more solutions quickly and to diversificate the solution space.

Bibliography

[1] M. Dell'Amico et al. "Mathematical models and decomposition methods for the multiple knapsack problem". In: *European Journal of Operational Research* 274.3 (1 May 2019), pp. 886–899. DOI: https://doi.org/10.1016/j.ejor.2018.10.043.

[2] CE. Ferreira, A. Martin, and R. Weismantel. "Solving Multiple Knapsack Problems by Cutting Planes". In: *SIAM Journal on Optimization* 6.3 (1996), pp. 858–877. DOI: https://doi.org/10.1137/S1052623493254455.

[3] S. Eilon and N. Christofides. "The Loading Problem". In: *Management Science* 17.5 (Jan. 1971), pp. 259–268. DOI: https://doi.org/10.1287/mnsc.17.5.259.

[4] E. Angelelli, R. Mansini, and MG. Speranza. *Kernel Search: a heuristic framework for MILP problems with binary variables.* Technical Report of the Department of Electronics for Automation R.T. 2007-04-56. University of Brescia, Dec. 2007.

[5] E. Angelelli, R Mansini, and MG. Speranza. "Kernel search: A general heuristic for the multi-dimensional knapsack problem". In: *European Journal of Operational Research* 37.11 (Nov. 2010), pp. 2017–2026. DOI: https://doi.org/10.1016/j.cor.2010.02.002.

[6] E. Angelelli, R Mansini, and MG. Speranza. "Kernel Search: a new heuristic framework for portfolio selection". In: *Computational Optimization and Applications* 51.1 (Jan. 2012), pp. 345–361. DOI: https://doi.org/10.1007/s10589-010-9326-6.