Soft Computing manuscript No.
(will be inserted by the editor)

Evolutionary-Based Tailoring of Synthetic Instances for the Knapsack Problem

Luis Fernando Plata-González · Ivan Amaya · José Carlos Ortiz-Bayliss · Santiago Enrique Conant-Pablos · Hugo Terashima-Marín · Carlos A. Coello Coello

Received: date / Accepted: date

Abstract The assessment of strengths and weaknesses of a solver is often limited by the diversity of the cases where it is tested upon. As such, it is paramount to have a versatile tool which finds the problem instances where such a solver excels/fails. In this manuscript, we propose to use an evolutionary algorithm for creating this tool. To validate our approach, we conducted several tests on four heuristics for the knapsack problem. Although, the process can be extended to other domains with relatively few changes. The tests cover different sets of instances, both favoring the performance of one heuristic while hindering that of the remaining ones, and vice versa. To further test our evolutionary-based model, we also apply it on a recent approach that combines the strengths of different heuristics to improve its performance (usually referred to as a hyper-heuristic). We show that it is possible to tailor instances in which even this more complex model excels/fails. Throughout our approach, a researcher can test a solver under different kinds of scenarios, delving deeper into the conditions that make it perform well/poorly. Therefore, we recommend using the proposed approach as a means to grasp better insights about strengths and weaknesses of different solvers.

Keywords Evolutionary computation · Knapsack problem · Instance generation

1 Introduction

The knapsack problem (KP) is a fundamental and extensively studied problem in combinatorial optimization, not only for its theoretical interest but also because of its many practical applications. This problem is classified as NP-hard, and several exact and approximate algorithms have been developed to solve it (Gao et al, 2014; Furini et al, 2017; Mavrotas et al, 2015). KP has multiple variants, as described in the works of Martello and Toth (1990) and Kellerer et al (2004).

Knapsack problems are important, among other things, because they can be used to model certain kinds of real-world problems (Zitzler and Thiele, 1999; Samavati et al, 2017), including budgeting, cargo loading and cutting stock (Martello and Toth, 1990). A recent example of the many applications of this problem is described in the work of Simon et al (2017), where a group of people must decide which items to carry and how to distribute them. The authors modeled their problem as multi-dimensional (d-KP) and multiple knapsack problems. Li et al (2016) illustrate another feasible scenario,

This research was supported in part by Consejo Nacional de Ciencia y Tecnología (CONACyT) Basic Science Project [grant number 241461] and ITESM Research Group with Strategic Focus in intelligent Systems. The last author gratefully acknowledges support from CONACyT project no. 221551.

L. F. Plata-González · I. Amaya (Corresponding Author) · J. C. Ortiz-Bayliss · S. E. Conant-Pablos · H. Terashima-Marín
Tecnologico de Monterrey, School of Engineering and Sciences, Ave. Eugenio Garza Sada 2501, Monterrey, N.L., Mexico, 64849
Tel.: +52 01 81 8358 2000
E-mail: iamaya2@tec.mx

C. A. Coello Coello
CINVESTAV-IPN (Evolutionary Computation Group), Ave. Instituto Politcnico Nacional 2508, Mexico City, Mexico, 07360

List of abbreviations: Mixed Integer Programming Library (MIPLIB), Default heuristic (Def), Max Profit heuristic (MaP), Max Profit per Weight heuristic (MPW), Minimum Weight heuristic (MiW), Latin Hyper-cube Sampling (LHS) and Hyper-heuristic (HH).

using a multiple knapsack model to solve the optimization problem of a sensor network. A third example, in yet another field, rests in the work of Szkaliczki et al (2014), where solutions related to the KP were introduced to study a problem related to streaming of layered video contents over peer-to-peer networks.

Finding new strategies for solving the knapsack problem, as well as other problems, requires a set of suitable instances where these methods can be tested. For example, Marinakis and Marinaki (2014) used two different benchmarks to test the performance of a metaheuristic on the open vehicle routing problem: a traditional one, proposed by Christofides et al (1979); and a more recent one, proposed by Li et al (2007). Similarly, Ariyasingha and Fernando (2015) tested different versions of the Multi-Objective Ant Colony Optimization (MOACO) algorithm on instances of the traveling salesman problem presented by Reinelt (1991).

Although the past years have witnessed a remarkable improvement in the quality of the solving methods, benchmark instances have remained almost unaltered. Having access to challenging instances may help researchers to identify critical areas where improvements to solvers can take place. In this regard, real-world scenarios are an ideal but scarce source of test problems. Nonetheless, another feasible source rests in the usage of synthetic instances.

Some of the instances that have been used as benchmarks in the knapsack problem include the work of Martello and Toth (1990). Here, the authors introduced a set of 0/1 KP instances with different numbers of items, and a high correlation between weights and profits of the items. Another important benchmark source is the OR-Library, which was created by Beasley (1990) for distributing test data for Operations Research (OR) problems, including the multiple knapsack problem. Azad et al (2014) recently used this library for assessing the performance of a metaheuristic in 0/1 multidimensional knapsack problems. Yet another example of benchmarks is presented in the work by Zitzler and Thiele (1999), where the authors generated data for the multi-objective KP, by using random integers for weights and profits. This dataset has been widely used in other works on evolutionary multi-objective optimization (Zitzler et al, 2001; Lust and Teghem, 2012; Knowles and Corne, 2000).

In (Martello et al, 2000), the generated instances were divided into different classes. Some of those instances contained specific ratios of different kinds of items, while others had items created by generating random numbers within specific ranges. For one specific class –"all-fill", as the authors named it– they used a recursive process to create instances starting from the solution and going all the way back to the items. Similarly, Pisinger (2005) produced several sets of instances, considered hard, by using a random generator that manipulates the correlation between profit and weight of the items. Another example corresponds to the Mixed Integer Programming Library (MIPLIB) (Koch et al, 2011), originally proposed in 1992. The current version dates back to 2010 and contains several types of domains, including KP. More recently, Petursson and Runarsson (2016) implemented an instance generator for a multidimensional KP. Although the authors included parameters such as the tightness ratio for controlling the properties of the generated instances, they still relied on a uniform distribution for the generation of the weights and profits of the items.

As mentioned before, synthetic instance generation for various types of KP instances has been explored in the past. However, in most of the approaches available in the literature, parameters such as the profit and the weight of items are based on randomly distributed values. Moreover, empirical evidence suggests that it may not be enough to generate instances with arbitrary properties, but to generate them in such a way that they can be used to obtain valuable information from the behavior of specific solvers. This cannot be done with the aforementioned techniques, since instance generation is independent from any solving strategy.

We consider that the discussed approach deals with the problem in a reactive way, by creating problems and then assessing if solvers were able to perform well/poorly. Thus, in this manuscript we propose a more proactive approach. Our idea considers an evolutionary-based model that finds instances in which the solvers exhibits a desired behavior. This approach may allow for future research about the strengths and weaknesses of said solvers.

This need for reliable test instances that exploit specific weaknesses of the solving methods is not new and has received an increasing attention in recent years. A recurring idea in different domains is to use an 'intelligent' generator based on evolutionary computation, creating models that facilitate the analysis of each method. This model has led to interesting results. A clear example found in the literature relates to constraint satisfaction problems (CSPs), where van Hemert (2003, 2006) showed how to use an evolutionary algorithm to detect hard to solve instances. His evolutionary algorithm maintained a population of binary CSPs. Their structure changed over time, and the genetic operators altered conflicting pairs between two values of two variables. The set of variables and their domains were kept unchanged. Smith-Miles et al (2010); Smith-Miles and van Hemert (2011) proposed an evolution-

ary algorithm for producing distinct classes of traveling salesman problem (TSP) instances that are intentionally easy or hard for certain algorithms. In their analysis, a comprehensive set of features was used to characterize the instances. After analyzing the performance of those algorithms over the set of newly created instances, the authors proposed a high-level algorithm for predicting the search effort, as well as the algorithm likely to perform best over a set of unseen instances, which exhibited high accuracy.

Even so, and to the best of our knowledge, the use of evolutionary computation to generate instances tailored for specific solvers of the KP remains unexplored. Based on this, we asked whether it was possible to use a genetic algorithm for producing synthetic 0/1 KP instances where one or more solvers outperform the others. By doing so, we can fill the knowledge gap regarding the generation of tailored KP instances, while providing a tool for creating test beds with increased/decreased difficulty for a particular kind of solver. Therefore, we developed an evolutionary model powered by a genetic algorithm. We used such a model within the KP, striving to generate sets of instances tailored to four different heuristics, as well as to a high-level solver that combines the four of them.

Research carried out in this work yields an important contribution: a reliable technique for generating KP instances that exploit specific weaknesses or strengths of particular solvers. We thoroughly explore the behavior of this approach by analyzing the performance of four traditional heuristics on a large number of 0/1 KP instances. To further this exploration, we implement the recently proposed high-level solver known as a hyper-heuristic, which combines other solvers for tackling each problem, and we test our generation technique on it. By doing so, we aim to generate both, (1) instances where the hyper-heuristic excels and single heuristics fail and (2) instances where the hyper-heuristic fails but single heuristics excel. This generation is performed to determine if the proposed approach can adapt to a more complex solution method.

The remainder of this paper is organized as follows. Section 2 describes the fundamental ideas that support this work. The evolutionary-based generator of KP instances is detailed in Section 3. Section 4 is reserved for presenting the methodology followed in this work. In Section 5, we present our experiments and main results, along with their discussion. Finally, the conclusions and overview of future work is presented in Section 6.

2 Fundamentals

The KP is formally defined as a set of n items, where each of these items has a profit p_j and a weight w_j, and a container with a capacity C (Kellerer et al, 2004). Solving a KP requires selecting a subset of items in such a way that their combined profit is maximized while their total weight remains under the container capacity, C. The KP can also be represented as a linear integer programming formulation, using equations (1)–(3).

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq C \tag{2}$$

$$x_j \in \{0,1\} \;\; j = 1, \ldots, n \tag{3}$$

There are many interesting variants of the KP but, for the sake of brevity, we omit a detailed discussion of them. The interested reader should refer to (Martello and Toth, 1990) and (Kellerer et al, 2004) for more in-depth information. Even so, the methods exposed herein are also applicable to those variants, with none or a few changes.

2.1 Instances Considered for this Work

Throughout this work, we used instances that fall under two categories: traditional ones, which were taken from the literature for comparison purposes; and tailored ones, which were generated following the model proposed in this manuscript. Such categories can be briefly described as:

Traditional instances: Refer to those widely available in literature and that can be used as a reference point for comparing our model. In this work, we selected some of the instances discussed by Pisinger (2005)[1]. The author generated such instances through a random model based on the correlation of the weights and profits of items within an instance.

Tailored instances. Relate to those generated in this work, following the proposed model. We generated several sets of instances with specific behaviors [2] (for details, please refer to Section 4). In this way, we can

[1] These instances can be downloaded from `http://www.diku.dk/~pisinger/`

[2] These instances, including their optimum, are available upon request.

isolate a given solver, providing insights for a better understanding of the conditions where it performs appropriately. It is worth remarking that the generated instances were created in such a way that they could be solved through dynamic programming, so that their global optimum can be known.

2.2 Instance Characterization

We require a set of features that allow us to characterize and understand the nature of a given instance. Thus, in this work we considered seven features, calculated over the set of unpacked items in the instance: $\bar{w}$ (mean weight value, divided by the maximum weight), $\widetilde{w}$ (median weight value, divided by the maximum weight), σ_w (standard deviation of the weights, divided by the maximum weight), $\bar{p}$ (mean profit value, divided by the maximum profit), $\widetilde{p}$ (median profit value, divided by the maximum profit), σ_p (standard deviation of the profits, divided by the maximum profit) and r (weight-profit correlation, divided by two and shifted upwards by 0.5). It is worth noting that all these features were selected for this work based on empirical findings.

The values that these features can take lie in the $[0, 1]$ range. For example, consider a subset of remaining items whose weights are $(2, 2, 3, 4)$ and whose profits are $(10, 5, 6, 15)$, respectively. Thus, each of the aforementioned features (in order) will take the values of $(0.69, 0.63, 0.24, 0.60, 0.53, 0.30, 0.69)$. Moreover, features are dynamic, i.e., they change as the instance is solved.

2.3 Available Solvers

In this work, we focused on two kinds of solvers: low-level heuristics, which are fast and simple but do not offer any way of adapting to problems; and high-level ones, known as hyper-heuristics, which strive to learn the behavior of a problem so they can predict how to best solve it. In the first case, we included four approaches that operate over the set of unpacked items in the KP instance: Default (Def), which selects the first available item; Max Profit (MaP), which selects the item with the largest profit; Max Profit per Weight (MPW), which selects the item with the largest profit over weight ratio; and Min Weight (MiW), which selects the item with the minimum weight.

As mentioned above, low-level heuristics do not adapt to a problem. Instead, they offer a straightforward, deterministic, and computationally inexpensive algorithm. Due to this behavior, they can perform well in some scenarios but poorly in others. For this reason, we decided that the second kind of solver was needed in order to test the instances generated with our model.

A recent approach for problem solving is the use of selection hyper-heuristics (Burke et al, 2013): algorithms for selecting algorithms. The idea behind these high-level solvers is that a problem can be characterized by a set of domain-specific features, so that each instance of the problem can be placed as a point within the hyper-space. Furthermore, hyper-heuristics benefit when the following happens: first, each instance can be solved best with one specific low-level heuristic (or with some combination of them), creating zones of action for each solver; second, such behavior can be mapped back to the features. Thus, a selection module can learn the pattern of a specific domain, creating a set of rules for dividing the hyper-space into zones where a given algorithm must be applied. There are different variants and kinds of hyper-heuristics, but a detailed description of them is beyond the scope of this manuscript. We will only mention that the type of hyper-heuristic we considered for this work begins solving a problem with a specific heuristic (e.g., Def), and may switch to another one later on (e.g., MiW). For more details, the interested reader is welcomed to consult (Burke et al, 2013), (Drake et al, 2014) and (Drake et al, 2016).

Throughout this work, we used hyper-heuristics that could select among these four single heuristics (Def, MaP, MPW, and MiW). We followed the hyper-heuristic model previously described in (Ortiz-Bayliss et al, 2016), where a messy genetic algorithm finds a set of rules that determine when to use one particular heuristic, based on a set of features that characterize the current problem state. The idea in this model is to minimize the error associated to the set of rules. Thus, at each iteration, the genetic algorithm evaluates a candidate solution (a set of rules) over a training subset of instances, and determines the corresponding error level. Using this information, the algorithm improves the population and moves on to the next iteration. At the end, the algorithm reports a set of rules which perform well over the training set, and that can be used for solving new instances. Details regarding the inner workings of the genetic algorithm are presented below.

2.4 Genetic Algorithms

Genetic algorithms (GAs) are an optimization tool inspired in how evolution occurs in nature (Goldberg, 1989; Holland, 1975). One element of particular interest in the nature of GAs is their competitive behavior, since members within the population are always struggling to determine who is worthy of progressing to the next phase of evolution. The generic version of a

steady state genetic algorithm works as follows. The algorithm produces a randomly initialized population of individuals (potential solutions or configurations) and runs for a given number of iterations. In our case, individuals in the population are represented as binary strings. The initial population is evaluated according to a fitness function. At each iteration, two individuals are randomly selected from the population and the one with the best fitness is selected as the first parent. This procedure is repeated to select a second parent. Both parents are then used for creating two offspring by using genetic operators (crossover and mutation). The offspring are appended to the population and the two individuals with the worst fitness are removed from the population. This process is repeated until a termination criterion is met. When the process is over, the GA returns the individual with the best fitness value.

3 The Evolutionary-Based Instance Generator

The generator proposed in this investigation relies on a GA that evolves the structure of potential KP instances, guided by a fitness function. The GA modifies the weight and profit of all items until it finds a configuration where the solvers exhibit the desired behavior.

To produce an instance, four arguments are required: the capacity of the knapsack, W; the number of items, n; the maximum weight per item, w; and the maximum profit per item, p. The process starts with a population of randomly initialized binary strings. From this point on we will refer to these binary strings as chromosomes.

Each chromosome encodes a KP instance of n items. Each item in the encoded instance is represented by $l_w + l_p$ bits, where $l_w = \lceil \log_2(w) - 1 \rceil$ and $l_p = \lceil \log_2(p) - 1 \rceil$. Therefore, the length of the chromosome, l, is given by $n \times (l_w + l_p)$. Figure 1 depicts an example of a chromosome generated with $n = 5$, $w = 4$, and $p = 8$, which encodes a KP instance of five items, with weights and profits of $(3, 1, 2, 1, 4)$ and $(8, 6, 4, 5, 2)$, respectively.

The chromosomes will change as the result of the evolutionary process. In this study, the number of items, as well as the capacity of the knapsack, are not encoded within the chromosome and then, they remain unaltered throughout the evolutionary process. However, the weight and profit of each item change as an attempt to reach a KP instance with the desired complexity. To do so, two new offspring are created at each iteration, using the genetic operators described below, and the two worst chromosomes are removed from the population.

3.1 Genetic Operators

For this investigation, we have considered three genetic operators: selection, crossover and mutation. The selection process involves a tournament selection of size TS. For each iteration, the algorithm randomly selects TS chromosomes and keeps the one with the highest fitness. The process is repeated to select a second parent. With a probability CR, the crossover operator is applied to the parents to produce two offspring. To perform crossover, one point (c_x) is randomly selected. The offspring will be formed according to the following strategy: the first offspring will contain the first c_x bits of the first parent and the last $l - c_x$ bits of the second one. Conversely, the second offspring will contain the first c_x bits of the second parent and the last $l - c_x$ bits of the first one. After crossover has occurred, mutation may be applied. With a probability MR, the mutation operator is applied to each of the offspring. When mutation takes place, one randomly selected position of the chromosome is flipped.

3.2 The Fitness Function

Throughout this work, we consider the generation of two kinds of instances: easy-to-solve and hard-to-solve. The generation, thus, focuses on favoring (or hindering) the performance of a given solver (in this case, one of the low-level heuristics from Section 2.3), relative to the remaining ones. Nonetheless, our decision does not limit the scope of the proposal, since the kind of instances generated mainly depends on the functions described herein. Therefore, different behaviors can be implemented so that instances with a different nature can be generated.

In the first case (easy-to-solve instances), the idea is that the selected solver outperforms the others with a gap as big as possible. Therefore, eq. (4) shows a feasible fitness function (FF), where Fit_{one} represents the profit of the selected solver on the KP instance, and Fit_{others} represents the fitness of the remaining solvers on the same instance. This implies that, at each iteration, all the solvers must run to calculate the quality of the current solution. Nonetheless, since each solver corresponds to a straightforward heuristic, this process is not computationally expensive. In the second case (hard-to-solve instances), the idea is that the selected solver performs badly compared against the other available solvers. So, the comparison must be inverted by striving to widen the gap between the selected solver and the worst one among the remaining ones. The current optimization problem requires a maximization of

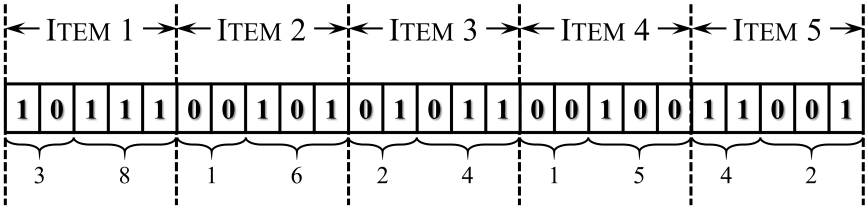


Fig. 1 A chromosome that encodes a KP instance of five items with weights and profits of $(3, 1, 2, 1, 4)$ and $(8, 6, 4, 5, 2)$, respectively.

the aforementioned differences. For this reason, equations (4) and (5) include a minus sign to make them suitable for optimizing through minimization of the fitness function (FF). Nonetheless, bear in mind that the optimization problem can be directly solved as a maximization problem.

$$FF = -\left(Fit_{one} - \max\left(Fit_{others}\right)\right) \tag{4}$$

$$FF = -\left(\min\left(Fit_{others}\right) - Fit_{one}\right) \tag{5}$$

4 Methodology

We divided our research into a three-stage work, as shown in Figure 2. A description of each stage is provided below. It is important to remark that, since each instance may have a different optimum, we opted for analyzing the quality of the solution relative to the best known solution (in this work, the optimum solution was obtained via dynamic programming). Therefore, results for the next section will range from zero to one, depending on how close they were from the known optimum.

4.1 Preliminary Tests

Throughout this stage, we focused on testing out the feasibility of our approach. Thus, experiments were designed to cover a wide range of behaviors, but at the same time keeping in mind that tests could not be exhaustive. Therefore, we generated instances for all heuristics with an arbitrary problem configuration (knapsack capacity of 50 and 20 items, where each item could have maximum profit and weight of 100 and 10 units, respectively), and with fixed parameters for the generator (mutation rate of 0.1, crossover rate of 1.0, population size of 10, and tournament size of 2). We ran 60 repetitions for each configuration and demanded instances that were easy (and hard) to solve with each heuristic. This yielded a total of 480 different instances (60 runs, 8 configurations), which were then solved with each heuristic to verify that each heuristic performed appropriately over their corresponding sets. Afterwards, we used Principal Component Analysis (PCA) over all the data (considering all seven features), creating a two-dimensional plot of the sets and trying to determine trends. Furthermore, we included the data from four reference sets, extracted from the literature (Pisinger, 2005) and which are considered to be hard-to-solve, and mapped them into the reduced feature space to verify their location relative to the one of recently generated instances.

4.2 Initial Testing

As a second stage, we decided to improve upon the parameters of the genetic algorithm. Therefore, we used different configurations and mapped the performance of our generation model for each of them, seeking to select a proper set of parameters for each problem scenario. Since our model is stochastic in nature, it must be repeated several times to determine an average performance. However, the idea is that this procedure can also be used for future research (so that parameters of the generation model can be tuned) and, therefore, its computational cost should not be too high. Hence, we decided to run the instance generator 30 times for each configuration. To calculate the average performance of the generator, many more instances must be generated. In fact, the number of runs of the genetic algorithm for this stage considerably increased.

We used latin hyper-cube sampling (LHS) to generate the set of parameter configurations. Assuming that four parameters of the genetic algorithm are to be selected (crossover rate, mutation rate, population size, and tournament size), that feasible ranges for each one are known, and that the number of points for each interval can be established, then the total number of parameter settings can be determined. For this work, we considered the parameter configurations given by Table 1, which result in a total of 3200 combinations. LHS was used to generate 160 parameter configurations (5%

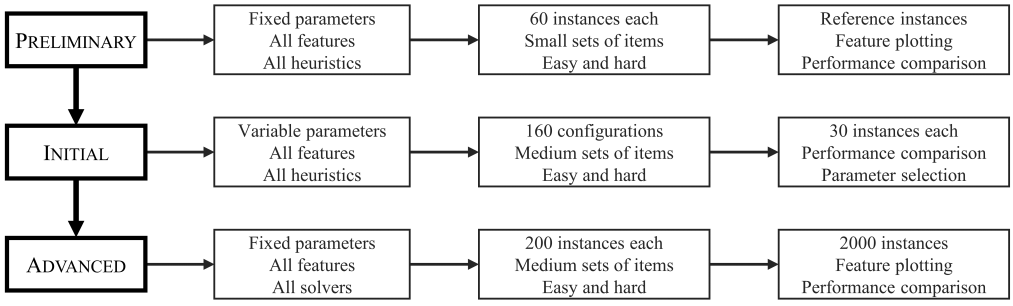


Fig. 2 Three-stage methodology followed throughout this work.

of the total number of available combinations), yielding a total of 38400 runs of the instance generator. These configurations are not shown due to space restrictions.

Table 1 Ranges and number of points considered for each parameter of the evolutionary strategy. The total number of combinations is 3200.

Parameter	Range	Number of Points
Population Size (PS)	[10, 150]	8
Crossover Rate (CR)	(0, 1.00]	10
Mutation Rate (MR)	(0, 0.02]	10
Tournament Size (TS)	[2, 5]	4

With the aim of delving deeper within the adaptive capabilities of the proposed model, for this stage we requested instances with the following problem characteristics: knapsack capacity of 25 and 40 items, with maximum profit of 100 and maximum weight of 20 each. We used, once again, PCA to reduce the number of dimensions from seven to two. Nonetheless, we carried out a new analysis since the configuration of the problem was altered.

4.3 Advanced Tests

The final stage of testing focused on exploring the scalability of the proposed instance generator. Thus, we selected the best configuration yielded by the previous stage, and used it to generate 200 instances per heuristic. Again, all eight scenarios were considered (four heuristics, with easy and hard instances for each one), so a total of 1600 instances were created. In this case, the problem configuration of the previous stage was preserved, as we strove to determine feature regions where each heuristic performs poorly and/or appropriately.

Aside from analyzing feature distribution across all instances (using PCA), in this stage we also analyzed the performance of a modern high-level solver. For this work, we decided to use a hyper-heuristic model able to select among all base heuristics (see Section 2.3). In theory, during the learning phase the solver must be able to identify which heuristic performs best for each scenario, so that a similar scenario in the test phase is solved appropriately. Moreover, we created 200 easy-to-solve and 200 hard-to-solve instances for the hyper-heuristic. This was done to analyze the location of instances that the hyper-heuristic can properly solve, by combining tools that are unsuited for the task. But, it was also done to analyze instances where a combination of appropriate tools leads to an unsatisfactory result.

As a last test, we selected two hard-to-solve problem configurations available in the literature (for 20 and 50 items), and generated easy and hard instances for each solver. Then, we plotted the main information of the generated instances against that of the ones used as a reference, and analyzed the performance of each solver over all sets (the ones generated and the ones selected as reference).

5 Results and Discussion

Following is a description of the results that we consider to be more relevant. Seeking to ease the readability of the manuscript, and to facilitate the link between methodology and results, we opted to create one section for each stage of the methodology.

5.1 Preliminary Tests

As mentioned in Section 4, the first step we took was to verify that the proposed strategy was, indeed, able to generate instances tailored to a given solver. To that end, we show the performance of each heuristic over every set of freshly generated instances (Figure 3). There

is one subplot for each targeted heuristic (highlighted bars), and each one contains two groups of bars: one focused on evolving easy-to-solve instances and one focused on evolving hard-to-solve instances. It is interesting to see that our model performs appropriately, being able to generate instances with a performance gap across solvers and either favoring or hindering them, according to what the user desires, all while using the same problem parameters.

It is also astonishing to note that, in some cases, the generated instances can be extremely difficult for a given heuristic. For example, for the Default heuristic, case (a), the average performance was lowered below the 0.05 mark, but the remaining ones had a much better performance (above 0.95). Nonetheless, some solvers are way more robust, making it easy to find instances where they outperform the competition, but hard to find ones where they perform poorly. An example of this idea is exhibited by the Max Profit per Weight heuristic, case (c), where its average performance for easy-to-solve instances was more than 0.50 higher than the second best heuristic, but whose average performance for hard-to-solve instances was only about 0.08 lower than the second worst heuristic.

In order to verify whether the instances we generate are different from those available in the literature, we selected four sets with different number of elements which are considered to be hard-to-solve, and plotted the performance of each heuristic over them. We can observe that the instances we generated allow for a better differentiation in the performance of the solvers (Figure 4). Thus, we have evidence suggesting that the proposed approach could be a fruitful path to pursue.

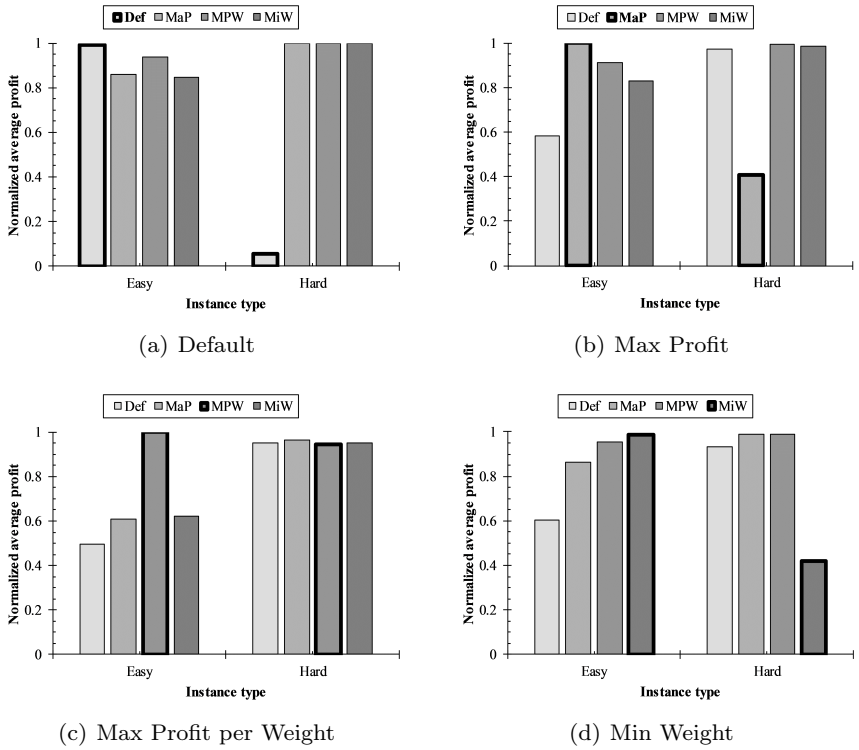
Since there are gaps in the performance of heuristics, this must imply that each heuristic performs best in some regions of the feature space, and worst in others. To verify this idea, we plotted the easy-to-solve and hard-to-solve instances generated throughout this stage for each heuristic (Figure 5). This leads to four images, where each one contains two clearly separated regions (one for each kind of problem). Figure 6 shows the centroids of each cluster from Figure 5 (and their distances to the origin), plus the centroid of a set of randomly generated instances and of the reference sets, for comparison purposes. It is worth remarking that these plots correspond to data yielded by PCA, which reduced information from the original seven dimensions (one for each feature) to two, considering all generated instances (one set per heuristic, plus the random set).

As can be seen from the figures, easy (star) and hard (cross) sets are apart from each other (for a given heuristic). This behavior holds even for the MPW heuristic (blue markers), whose performance gap was the smallest one (see Figure 4). Another interesting element is the fact that easy-to-solve instances (for each heuristic) are harder to differentiate than hard-to-solve ones (see Figure 6), even though in all cases the desired behavior was achieved. The behavior of the reference sets is also amusing, as they were all located in virtually the same spot in the reduced feature space. This corroborates the fact that all heuristics performed similarly over them (see Figure 4).

5.2 Initial Testing

As defined in the methodology, for this batch of tests we changed the problem configuration, striving to test our proposed approach on larger instances. Also, we ran tests with several configurations, though feature plots are omitted for the sake of brevity. Figure 7 shows a summary of the resulting profits, averaged across all repetitions for the best and worst configurations. In all cases, our proposed approach achieved its objective (positive deltas in the plot). Even so, there is an interesting behavior in the data, and it is the fact that it was quite easy/hard to generate instances for some of the scenarios. Furthermore, whenever it is straightforward to generate easy-to-solve instances for a given heuristic, it becomes difficult to find them for the opposite case (hard-to-solve instances for the same heuristic). It is precisely under these conditions that properly selecting the solver parameters becomes crucial. As an example, consider the instances generated for the MPW heuristic. Easy-to-solve instances can be found without too much hassle, and the difference between the averaged performance of MPW and the best one of the remaining solvers escalates to about 0.8 and 0.5; in the best and worst scenarios, respectively. However, evolving hard-to-solve instances become troublesome, lowering the difference to around 0.2 in the best case scenario. Furthermore, selecting a bad configuration for the solver makes it almost impossible to evolve an instance where MPW performs worst.

Motivated by these results, we ran a set of tests to verify whether the performance was dependent on the problem parameters. We used the following problem configuration: knapsack capacity of 100 and 100 items, with maximum profit of 100 and maximum weight of 100 each. Data show interesting behaviors (Figure 8). The first thing to note is that, again, it is extremely straightforward to find easy/hard instances for some heuristics and that scenarios have opposite behaviors. For example, consider the generation process of hard-to-solve instances for the Def heuristic: even with the worst configuration, the genetic algorithm is capable of producing instances whose normalized average profit



(a) Default (b) Max Profit

(c) Max Profit per Weight (d) Min Weight

Fig. 3 Normalized average profit for heuristics operating over easy and hard instances, generated through the evolutionary approach and for each heuristic (preliminary test stage). Highlighted bars represent the heuristic instances they were tailored for.

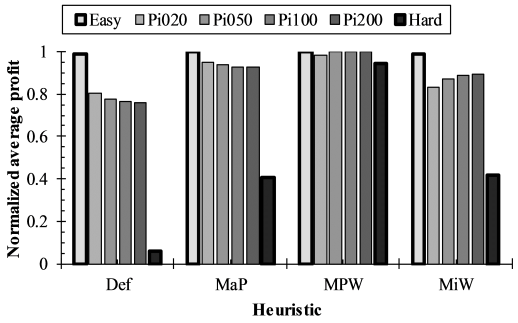
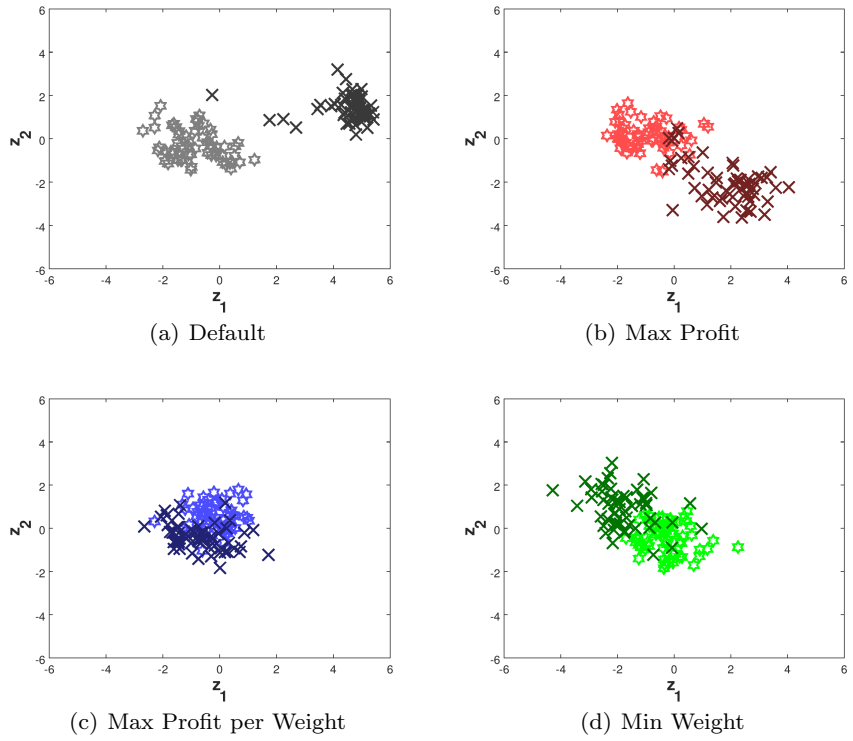


Fig. 4 Normalized average profit for all heuristics over their corresponding easy and hard instances, and over each reference set (preliminary test stage). Def: Default, MaP: Max Profit, MPW: Max Profit per Weight, MiW: Min Weight. Highlighted bars represent the heuristic instances they were tailored for.

differed by more than 0.99 from the closest heuristic. Nonetheless, the opposite scenario (generating easy-to-solve instances for Def) was not as effortless, and achieving the desired behavior depended on properly selecting a parameter configuration for the solver (negative values in Figure 8 indicate that behavior was not as desired). Moreover, if the problem parameters are not adequate (as in this example) it becomes impossible for the algorithm to perform appropriately in some scenarios, even at its best configuration (e.g., for MPW, where in all cases a negative delta was achieved).

Table 2 shows the best and worst configurations that were found at this stage (and used to generate the data from Figure 8). The biggest change in parameters was for MPW, where population size (PS) and mutation rate (MR), for hard-to-solve instances, are a sixth and a third of their counterpart, respectively. However, there were also some global patterns. In all cases there was a tendency for selecting more individuals for each tournament, and mutation rate (MR) was significantly lower for easy-to-solve instances than for their counterpart (except for MPW, where the be-



(a) Default (b) Max Profit

(c) Max Profit per Weight (d) Min Weight

Fig. 5 Easy (light-colored stars) and hard (dark-colored crosses) instances generated through the evolutionary approach for each heuristic, during the preliminary test stage.

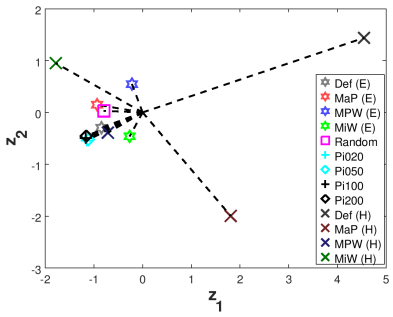
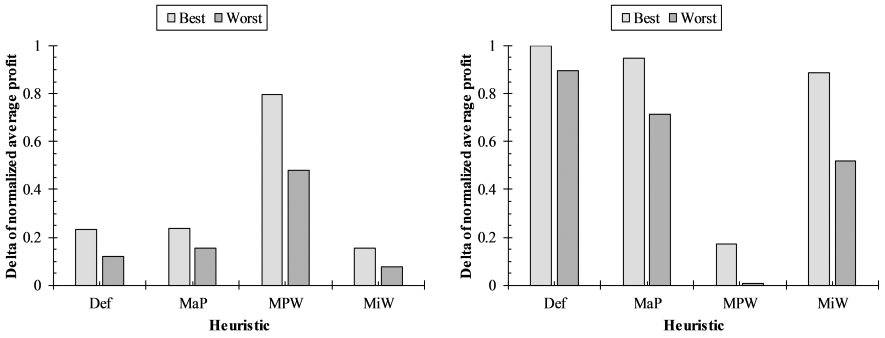


Fig. 6 Centroids of generated easy (E) and hard (H) instances for all heuristics during the preliminary testing stage, and their distance to the origin. Stars indicate easy sets and crosses represent hard ones. Random and reference (Pi) instances are included for comparison purposes. Def: Default, MaP: Max Profit, MPW: Max Profit per Weight, MiW: Min Weight.

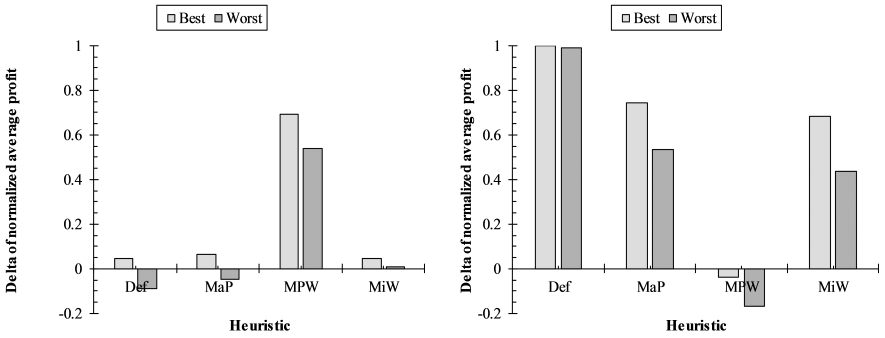
havior was the opposite). The crossover rate (CR), for easy-to-solve instances, was similar for Def, MaP and MPW (around 0.73), and quite different than that of MiW (0.238). However, in hard-to-solve instances MaP becomes the outcast (with a value of 0.23), while the remaining heuristics exhibited a similar rate (around 0.67). It certainly is interesting to see that CR focuses around these points.

At this point, we deem important to take a moment for glancing at why this might happen. Increasing the number of items, N_i, exponentially increases the number of combinations, N_{comb}, that must be explored by the evolutionary approach, as shown in eq. (6), where N_W and N_P are the number of feasible weights and profits for each item, respectively. Conversely, the number of items that can be included within the knapsack is limited by the remaining parameters of the problem, which can revert the effect of increasing the number of items to some extent. These bounds are shown in eqs. (7) and (8), where $N_{i_{min}}$ and $N_{i_{max}}$ are the minimum and maximum number of feasible items, respectively; C_{KP} is the capacity of the knapsack, while W_{min} and W_{max} are the minimum and maximum weight of the items. Even so, items that do not fit still need to take a value. Moreover, it could be the case that a given heuristic could take some items while the other ones could take another subset. We think that having more options to choose from (more items) makes it easier for all heuristics to easily find a relatively good solution. Thus, for harder scenarios (such as MPW) it is necessary to constrain the problem more, so that valid instances can be found. This makes it evident that there is a relation between the parameters of the problem (the number of items, the knapsack capacity and the max-



(a) Easy-to-solve instances (b) Hard-to-solve instances

Fig. 7 Best and worst deltas of normalized average profit for easy and hard instances for all heuristics, and while using an appropriate problem configuration. Def: Default, MaP: Max Profit, MPW: Max Profit per Weight, MiW: Min Weight.



(a) Easy-to-solve instances (b) Hard-to-solve instances

Fig. 8 Best and worst deltas of normalized average profit for easy and hard instances for all heuristics, and while using an inappropriate problem configuration. Def: Default, MaP: Max Profit, MPW: Max Profit per Weight, MiW: Min Weight. Negative values indicate that behavior was not as desired.

Table 2 Best and worst configurations for each heuristic, considering easy (E) and hard (H) to solve instances. Def: Default, MaP: Max Profit, MPW: Max Profit per Weight, MiW: Min Weight. PS: Population size, CR: Crossover rate, MR: Mutation rate, TS: Tournament size.

	Best				**Worst**			
Type	**PS**	CR	MR	**TS**	**PS**	CR	MR	**TS**
Def (E)	36	0.772	0.065	5	41	0.317	0.001	3
MaP (E)	139	0.720	0.047	3	123	0.368	0.109	3
MPW (E)	157	0.709	0.153	4	64	0.851	0.095	4
MiW (E)	47	0.238	0.032	4	49	0.837	0.020	4
Def (H)	30	0.650	0.173	5	41	0.317	0.001	3
MaP (H)	95	0.230	0.169	4	48	0.350	0.020	2
MPW (H)	25	0.636	0.051	3	38	0.051	0.156	2
MiW (H)	99	0.709	0.153	4	62	0.126	0.143	2

imum weight, and perhaps, profit). However, this relation was not further studied in this manuscript since the focus was to develop an approach to generate instances with a specific goal in mind (favoring or hindering a given heuristic). Nonetheless, it would certainly be interesting to investigate further into this phenomenon, and using a tool such as the one presented here could prove fruitful for running tests with specific behaviors.

$$N_{comb} = (N_W \cdot N_P)^{N_i} \tag{6}$$

$$N_{i_{min}} = \frac{C_{KP}}{W_{max}} \tag{7}$$

$$N_{i_{max}} = \frac{C_{KP}}{W_{min}} \tag{8}$$

5.3 Advanced Testing

Since the parameters of the problem (knapsack capacity, number of items, maximum profit, and maximum weight) changed for the second part of this manuscript, for the third stage of this research it was necessary to perform a new PCA on the features that characterize the instances. Thus, a set of 1800 problem instances (the 1600 generated for this section, plus new 200 random instances) were analyzed and mapped, yielding Figure 9. Once again, hard-to-solve instances (dark-colored crosses) were easier to differentiate than their counterpart (light-colored stars), whilst randomly generated instances (magenta square) were located near the origin of the transformed space. Nonetheless, each subset rendered the desired behavior in all cases (Figure 10). Once again, the difference between the performance of MPW and the other heuristics (when considering hard-to-solve instances for MPW) was not as big as in other scenarios.

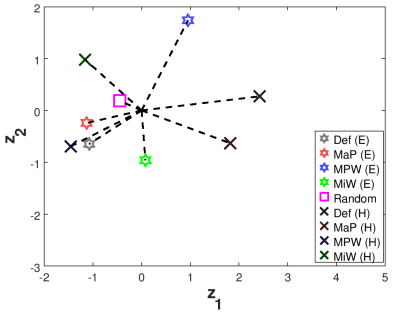


Fig. 9 Centroids of generated easy (E) and hard (H) instances for all heuristics during the advanced testing stage, and their distance to the origin. Stars indicate easy sets and crosses represent hard ones. Random instances are included for comparison purposes. Def: Default, MaP: Max Profit, MPW: Max Profit per Weight, MiW: Min Weight.

An interesting behavior that surfaces once more is the trade-off between easy-to-solve and hard-to-solve instances, since whenever it is easy to achieve big gaps in one of them, it becomes difficult to achieve them for the other one. But, perhaps more interesting is the behavior of MPW. This heuristic was the only one for which it was easier to create easy-to-solve instances, thus leading to wider gaps of performance (when compared against the remaining solvers). We consider that this can be due to the fact that this is the only heuristic that combines two metrics (profit and weight), whilst the other ones only consider one at a time (order, profit, or weight), or maybe because MPW is a more general purpose heuristic that works well under a broader set of scenarios. We deem necessary a deeper analysis of these ideas, which was not done here since it deviated from the focus of the manuscript.

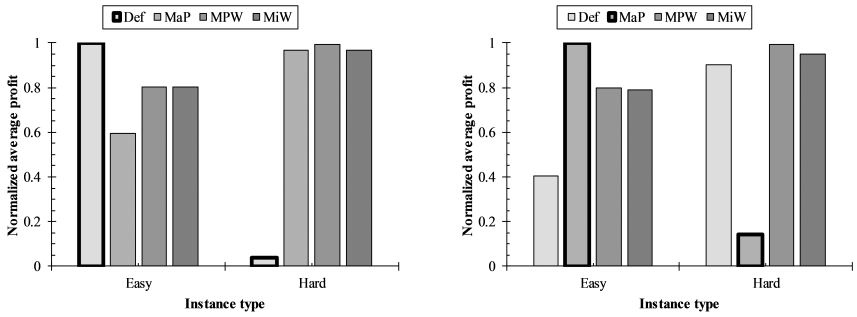
It is also alluring to analyze that, because hard instances for one solver can be rather easy for another one, it is common that they share some portions of the feature range. Moreover, and due to the way in which PCA works, it is also possible that the easy and hard-to-solve regions for a particular heuristic overlap, at least in some parts (Figure 11). Nonetheless, one of the main ideas that can be extracted from these maps is that our proposed approach generates instances whose features are located in a zone where the selected solver performs as desired. A second one is that each solver exhibits a different behavior when migrating from easy to hard instances. In the case of Def and MaP the movement goes from left to right, upwards and downwards, respectively. In the case of MPW and MiW the movement goes from right to left, downwards and upwards respectively. We think that a deeper study of this phenomenon could help in designing new features for the KP.

5.3.1 Behavior of a High-level Solver

As Figure 2 suggested, the idea in this final stage of testing is to analyze the behavior of a high-level solver operating over the 1600 instances we generated. Thus, we trained three different hyper-heuristics over half of the instances and compared their performance against that of the basic heuristics, for the remaining instances. We observed that, by using a hyper-heuristic approach, the performance gap existing between the best heuristic and an ideal selector could be reduced by more than 20% in all three cases.

We wanted to go a bit further in our testing, so we used our approach to generate easy and hard instances for the high-level solver. It is worth remarking that, for these tests, we used the best already trained solver and focused on generating instances for it instead of training a new solver at each iteration, which could lead to some kind of race condition between both evolutionary approaches. Nonetheless, a careful analysis of such scenario seems indeed interesting, and it could be studied in a future work.

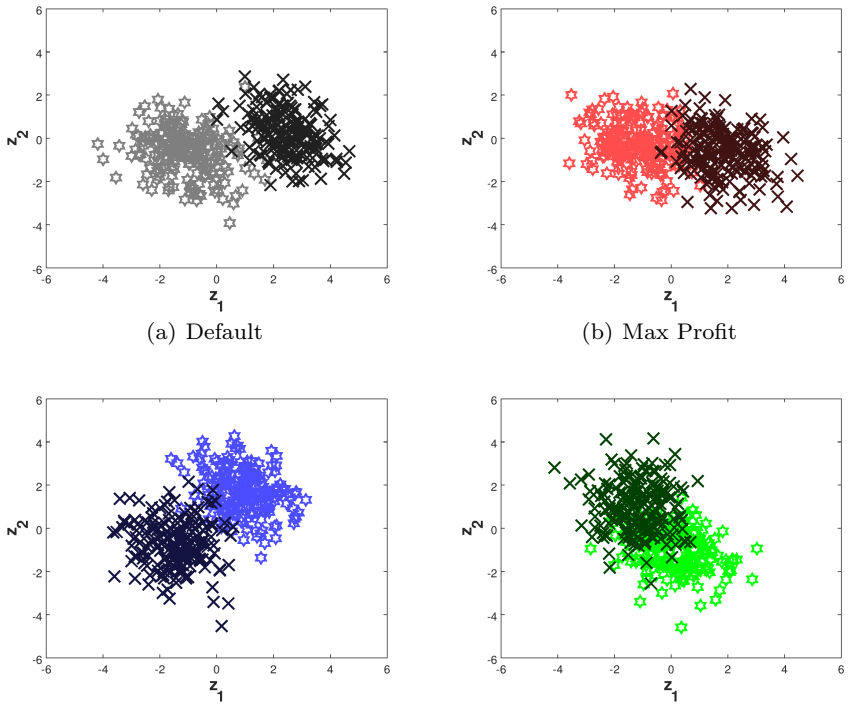
Using our evolutionary approach, we developed instances where the best single heuristic was, on average, 15% below the theoretical solution while the high-level solver was virtually perfect. This information is available on Figure 12(a). In fact, such a hyper-heuristic was able to find the theoretical optimum for 196 instances (i.e. 98% of the instances). On the other hand, an ideal heuristic selector (i.e. an Oracle that is able to perfectly predict the best heuristic for each instance) was unable



(a) Default (b) Max Profit

(c) Max Profit per Weight (d) Min Weight

Fig. 10 Normalized average profit for heuristics operating over easy and hard instances, generated through the evolutionary approach and for each heuristic (advanced testing stage). Highlighted bars represent the heuristic instances they were tailored for.



(a) Default (b) Max Profit

(c) Max Profit per Weight (d) Min Weight

Fig. 11 Easy (light-colored stars) and hard (dark-colored crosses) instances generated through the evolutionary approach for each heuristic, during the advanced testing stage.

to perfectly solve even a single instance. This means that no single heuristic is good at solving any of those instances, while combining them leads to finding a perfect solution most of the time. In fact, the average normalized profit for the Oracle sat at 0.8473, whilst that of the hyper-heuristic sat at 0.9994, meaning that the latter yielded about 15% more of the theoretical profit than the former. We also evolved, at will, sets where the opposite happened, having a heuristic with virtually perfect performance (i.e. MPW, which was able to find an average profit of 98.8% of the theoretical one.), but where the high-level solver fell 50% short, on average, of the best possible profits.

We also noted an interesting behavior of instances, since the migration from easy-to-solve instances to hard-to-solve ones implied a movement from right to left in the reduced feature space (as with MPW and MiW). However, as seen in Figure 12(b), there was almost no movement in the vertical direction. Instead, hard-to-solve instances distributed themselves over a more elongated region, as if trying to circle around easy-to-solve instances. We analyzed the distribution of features for each subset of instances, and we found that only the first three, which were related to the profit, varied. This indicates that by changing the profit of items within instances, it can become easier (or harder) for the high-level solver to behave properly.

As a final test, we analyzed whether our proposed approach was able to improve upon some of the instances proposed in (Pisinger, 2005). Therefore, we selected two problem configurations and demanded 100 easy-to-solve and 100 hard-to-solve instances for each solver. Again, our model evolves appropriately and delivers sets of instances where each solver excels and where each solver loses. Figure 13 summarizes this performance, and also shows the performance of each solver for the original set of instances. It is necessary to clarify that a definition of 'difficulty' is subjective and depends on the desired effect. As such, we are measuring the difficulty as the inability of a solver for finding the optimum solution. But, it could also be measured as the average time it takes to solve an instance with a given solver. Another interesting element in Figure 13 is the fact that both problem configurations yielded similar performance across all heuristics. This, in fact, maps back to the features we have considered in this work, as each set of instances clusters in a similar region for both problem configurations (see Figure 14).

Moreover, the fact that the performance of the original set of instances sits between easy-to-solve and hard-to-solve instances is due to an overlap in the feature space, where the original set has instances located in both, easy and hard regions of each solver (see Figure 15). Nonetheless, there is a clearer separation between both subsets of instances as the performance gap between them increases (e.g., for Default and Max Profit). It is worth mentioning that, due to space restrictions, only instances for the problem configuration with 50 items are plotted.

5.4 An example about how to transfer these ideas to other domains

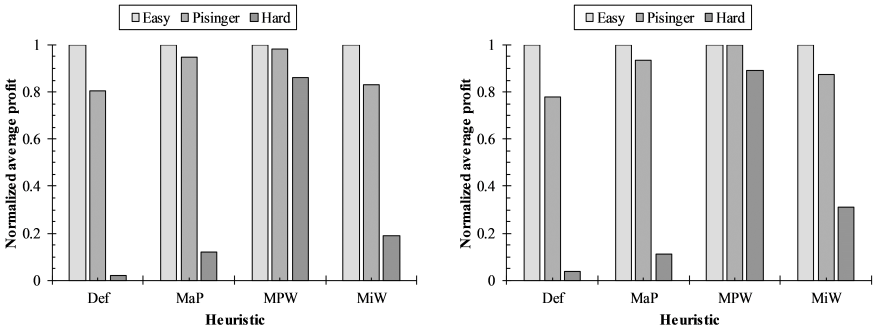
Striving to clarify the way in which the ideas presented in this work can be extended to other domains, we offer this bonus section. As we mentioned in section 3.2, equations (4) and (5) are only a couple examples of the behavior that can be generated with the instances. Thus, if we change those equations by equations (9) and (10), we can generate instances where solvers perform diversely (thus making selection a critical process), and instances where all solvers perform equally (rendering selection useless). We have explored this idea for Bin Packing Problems and achieved interesting results (Figure 16). Nonetheless, it is important to remark that equations (4) and (5) can also be used in this new domain. The only change that needs to be implemented is to replace Fit_{one} and Fit_{others} by the fitness yielded by the solvers from the Bin Packing Problem domain. However, we do not want to extend too much here, so we invite the reader to consult Amaya et al (2018).

$$FF = -\sqrt{\frac{\sum_{i=1}^{N_H}(Fit_i - Fit_{avg})^2}{N_H - 1}} \tag{9}$$

$$FF = (Fit_{Best} - Fit_{Worst})^2 \tag{10}$$

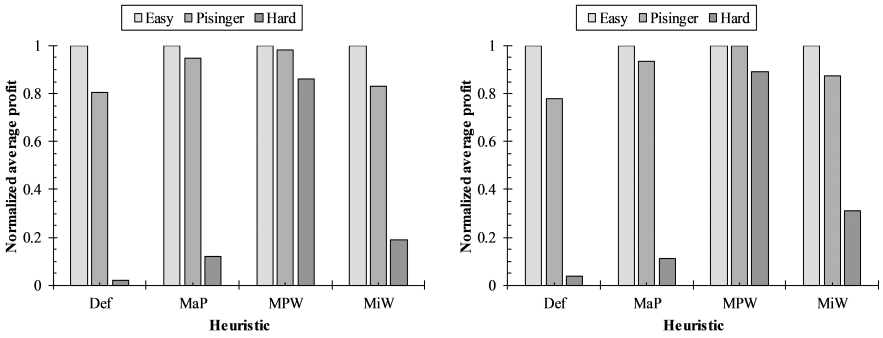
6 Conclusion and Future Work

Throughout this work we focused on using an evolutionary approach as the cornerstone for creating a tool that may help in better understanding solvers from a particular problem domain. In particular, our proposed tool can generate easy-to-solve and hard-to-solve instances for a given heuristic. This decision, though, does not limit the generality of our approach, since other objective functions can be used for evolving instances with diverse behaviors (see Section 3.2). In the first case, i.e. easy-to-solve instances, our tool provided instances where the desired solver virtually reached the theoretical optimum. In fact, the worst case scenario was for



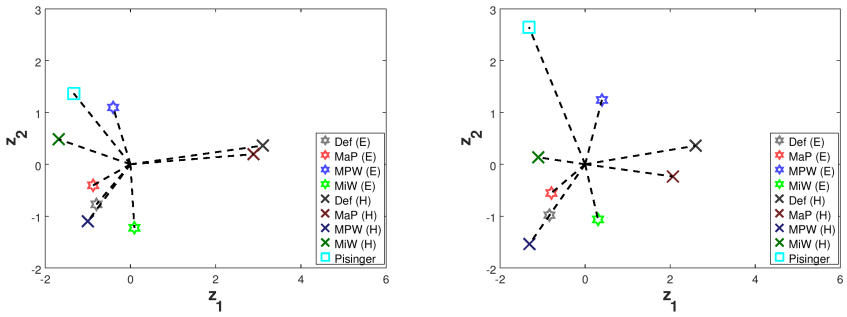
(a) Performance of each solver (b) Distribution of instances

Fig. 12 Left: Normalized average profit for all solvers operating over easy-to-solve and hard-to-solve instances tailored to the hyper-heuristic model. Right: Easy (light-colored stars) and Hard (dark-colored crosses) instances generated through the evolutionary approach for an already trained high-level solver. Def: Default, MaP: Max Profit, MPW: Max Profit per Weight, MiW: Min Weight, HH: Hyper-heuristic. Highlighted bars represent the solver instances they were tailored for.



(a) Configuration with 20 items (b) Configuration with 50 items

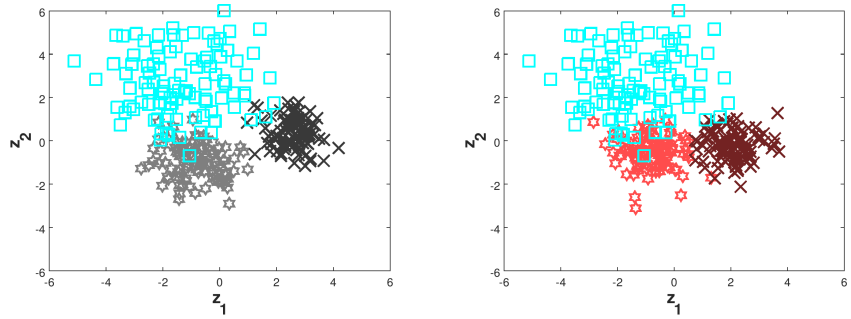
Fig. 13 Normalized average profit for all heuristics over their corresponding easy and hard instances, and over the reference sets with 20 and 50 items. Def: Default, MaP: Max Profit, MPW: Max Profit per Weight, MiW: Min Weight.



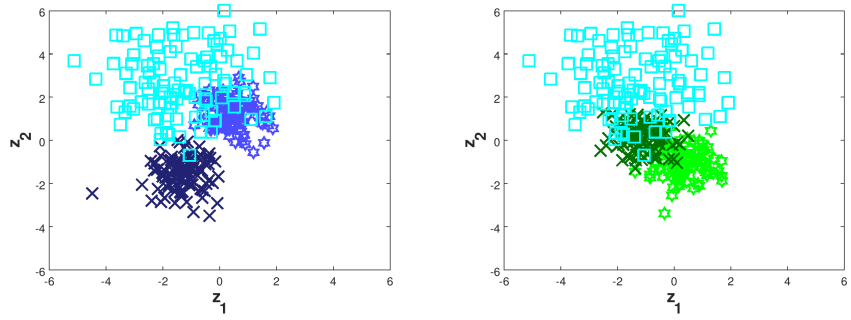
(a) Set with 20 items (b) Set with 50 items

Fig. 14 Centroids of generated easy (E) and hard (H) instances for all heuristics, departing from two problem configurations (available in the literature) with different numbers of items (cyan squares), and their distance to the origin. Stars indicate easy sets and crosses represent hard ones. Def: Default, MaP: Max Profit, MPW: Max Profit per Weight, MiW: Min Weight.

the Min Weight (MiW) heuristic, which yielded an average profit of 98.9% of the theoretical profit (calculated through dynamic programming). In the second case, i.e. hard-to-solve instances, our tool exhibited a more varied performance. The best case scenario was for the Default (Def) heuristic, which was only able to achieve a 5.7% of the maximum theoretical profit. The worst case scenario, however, was for the Max Profit per Weight heuristic, where we could only hinder performance in little more than 5% of the maximum theoretical profit.

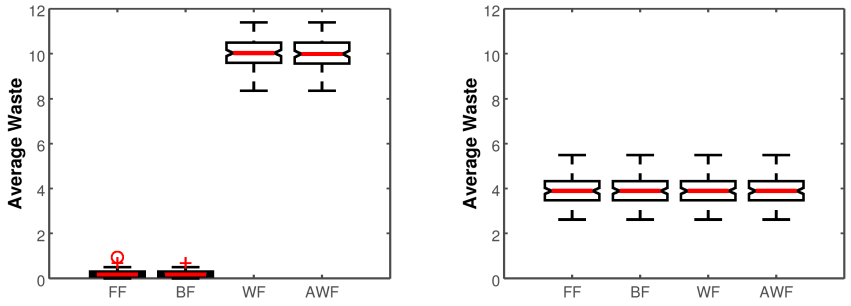


(a) Default (b) Max Profit



(c) Max Profit per Weight (d) Min Weight

Fig. 15 Easy (light-colored stars) and Hard (dark-colored crosses) instances generated through the evolutionary approach for each heuristic, departing from a problem configuration available in the literature (cyan squares) and considering 50 items.



(a) High Variation (b) Low Variation

Fig. 16 An example of the results that can be achieved in the Bin Packing Problem domain. Data shown represent the average waste achieved by all solvers when operating over instances generated with high variation (a) and with low variation (b), Amaya et al (2018).

Features are not required by our proposed approach, even though they could be used in case the solver requires them (e.g., in the case of hyper-heuristics). Even so, we did use seven different features for analyzing the behavior of each set of generated instances. We found interesting patterns. For example, in the case of the Default (Def) and Max Profit (MaP) heuristics, increasing the difficulty of instances implied a shift from the left side of the feature space to the right one, as well as upwards and downwards displacements (respectively). However, for the two remaining heuristics, the migration implied a shift in the opposite direction (i.e., from right to left), as well as a steeper vertical displacement (downwards for Max Profit per Weight and upwards for Min Weight). What seems more interesting about these patterns is that they remained even after changing the problem configuration to two previously reported ones. We deem as relevant an in-depth study of this phenomenon (see Section 5.3), e.g., by using the approach

we proposed in this work to generate instances with different levels of difficulty for each heuristic.

After testing our generation model with problem configurations readily available in the literature, we noticed that it was able to improve upon the already published data. For the two examples we selected, we evolved instances where each solver became virtually perfect (they found the theoretical optimum in all tested cases). We also evolved instances where most solvers were unable to yield a profit higher than 30% of the theoretical value (the only exception being for the Max Profit per Weight heuristic, where performance was hindered in about 15%). These data are certainly interesting, since this implies that our generation model can be coupled with previously published ideas to further improve the instances.

We also noticed that certain problem configurations undermined the evolution of instances with the desired behavior (even after selecting a suitable solver configuration). For example, we observed that capacity of the knapsack was an important factor for achieving hard instances for the Max Profit per Weight (MPW) heuristic, since it modified the relationship between the number of items to be evolved and the maximum number of items that could be packed, hence affecting the shape of the objective function. Thus, carrying out an in-depth study of parameter dependence is paramount for better understanding the knapsack problem. Furthermore, having a tool such as the one presented in this work could make the task an easier and more structured process, so we recommend using this approach for such an endeavor. As an illustrative case of what could be done in a future work, the reader may consider modifying the objective function to force instances with specific parameter ratios, or allowing for the evolutionary algorithm to directly optimize the capacity, or even allowing for some kind of co-evolutionary scheme. Another important avenue for future work, besides expanding the generator to other domains, resides in furthering the study of scalability for the presented generator, by creating considerably larger instances (with thousands of items). Likewise, the generator can be extended to include different versions of the knapsack problem, e.g., by considering multiple knapsacks and/or multiple objectives.

In light of the recent literature, thinking about high-level solvers most of the time implies dealing with features of a specific problem domain. Therefore, the tool presented herein could also be used to improve the performance of high-level solvers and advance that field, by generating instances with specific behaviors and with given feature values. This will facilitate an analysis of the efficiency with which features can map instances to specific solvers, and will undoubtedly lead to a better understanding of the elements that make a given heuristic perform well or poorly. For example, we evolved instances where a high-level solver was able to perfectly solve 98% of the instances, even though the base heuristics that composed it were unable to perfectly solve any instance at all. Thus, it is evident that there is a benefit from dynamically combining different solvers for tackling each instance. Moreover, the tool presented here can be extrapolated to other domains, so their benefits could be ripped across different optimization problems.

Our approach exhibits some points in favor, but it also has some drawbacks. Regarding the former, we have shown that our model successfully tailors instances that enhance or lessen performance of a given solver. But, it does not stop there. Our approach can evolve instances for more complex solvers. As an example, we targeted hyper-heuristics, which combine different solvers into a single one. Additionally, since we use an evolutionary approach for tailoring instances, users can provide different objective functions, customizing instances with different behaviors. Another benefit of using our approach is that different genetic operators can be used to expand its benefits. Also, our approach can be easily expanded to other problem domains. Regarding drawbacks, they mainly arise from not including the knapsack capacity nor the number of items into the encoding of the chromosome. They also cover the fact that the user cannot decide the degree of difficulty that an instance exhibits (though this could be solved by changing the objective function). Nonetheless, we made these decisions because they simplified the approach. We also considered that, by tailoring instances this way, it may be easier to arrive at conclusions about the influence that profit and weight hold over each solver. Future works should focus on improving these drawbacks so that we can provide a more robust tool.

Compliance with Ethical Standards

Funding: This study was funded by Consejo Nacional de Ciencia y Tecnología (CONACyT) Basic Science Project (grant numbers 241461 and 287479) and by ITESM Research Group with Strategic Focus in intelligent Systems. C.A. Coello Coello gratefully acknowledges support from CONACyT grant no. 2016-01-1920 (Investigacion en Fronteras de la Ciencia 2016).

Conflict of Interest: Luis Fernando Plata-González declares that he has no conflict of interest. Ivan Amaya declares that he has no conflict of interest. José Carlos Ortiz-Bayliss declares that he has no conflict of interest. Santiago Enrique Conant-Pablos declares that he has no conflict of interest. Hugo Terashima-Marín declares

that he has no conflict of interest. Carlos A. Coello Coello declares that he has no conflict of interest.

Ethical approval: This article does not contain any studies with human participants or animals performed by any of the authors.

References

Amaya I, Ortiz-Bayliss JC, Conant-Pablos SE, Terashima-Marín H, Coello Coello CA (2018) Tailoring Instances of the 1D Bin Packing Problem for Assessing Strengths and Weaknesses of Its Solvers. In: Auger A, Fonseca CM, Lourenço N, Machado P, Paquete L, Whitley D (eds) Parallel Problem Solving from Nature PPSN XV, Lecture Notes in Computer Science, vol 11101, Springer International Publishing, Cham, pp 373–384, DOI 10.1007/978-3-319-99259-4_30, URL http://link.springer.com/10.1007/978-3-319-99253-2http://link.springer.com/10.1007/978-3-319-99259-4{_}30

Ariyasingha IDID, Fernando TGI (2015) Performance analysis of the multi-objective ant colony optimization algorithms for the traveling salesman problem. Swarm and Evolutionary Computation 23:11–26, DOI 10.1016/j.swevo.2015.02.003, URL http://dx.doi.org/10.1016/j.swevo.2015.02.003

Azad MAK, Rocha AMAC, Fernandes EMGP (2014) Improved binary artificial fish swarm algorithm for the 0-1 multidimensional knapsack problems. Swarm and Evolutionary Computation 14:66–75, DOI 10.1016/j.swevo.2013.09.002

Beasley J (1990) OR-library: Distributing test problems by electronic mail. The Journal of the Operational Research Society 41(11):1069–1072

Burke EK, Hyde M, Kendall G, Ochoa G (2013) Hyper-heuristics : A survey of the state of the art. Journal of the Operational Research Society 64(12):1695–1724

Christofides N, Mingozzi A, Toth P (1979) The vehicle routing problem. In: Combinatorial Optimization, Wiley, pp 315–338

Drake JH, Hyde M, Ibrahim K, Ozcan E (2014) A genetic programming hyper-heuristic for the multidimensional knapsack problem. Kybernetes 43(9/10):1500–1511, DOI 10.1108/K-09-2013-0201

Drake JH, Özcan E, Burke EK (2016) A case study of controlling crossover in a selection hyper-heuristic framework using the multidimensional knapsack problem john. Evolutionary Computation 24(1):113–141

Furini F, Ljubić I, Sinnl M (2017) An effective dynamic programming algorithm for the minimum-cost maximal knapsack packing problem. European Journal of Operational Research 262(2):438–448

Gao J, He G, Liang R, Feng Z (2014) A quantum-inspired artificial immune system for the multiobjective 0-1 knapsack problem. Applied Mathematics and Computation 230:120–137

Goldberg DE (1989) Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley

van Hemert JI (2003) Evolving binary constraint satisfaction problem instances that are difficult to solve. In: Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC'03), IEEE Press, pp 1267–1273

van Hemert JI (2006) Evolving combinatorial problem instances that are difficult to solve. Evolutionary Computation 14(4):433–462

Holland JR (1975) Adaptation in Natural and Artificial Systems. The University of Michigan Press

Kellerer H, Pferschy U, Pisinger D (2004) Knapsack problems, vol 1. Springer, New York;Berlin;

Knowles JD, Corne DW (2000) Approximating the nondominated front using the pareto archived evolution strategy. Evolutionary Computation 8(2):149–172

Koch T, Achterberg T, Andersen E, Bastert O, Berthold T, Bixby RE, Danna E, Gamrath G, Gleixner AM, Heinz S, Lodi A, Mittelmann H, Ralphs T, Salvagnin D, Steffy DE, Wolter K (2011) MIPLIB 2010. Mathematical Programming Computation 3(2):103–163, DOI 10.1007/s12532-011-0025-9, URL http://mpc.zib.de/index.php/MPC/article/view/56/28

Li F, Golden B, Wasil E (2007) The open vehicle routing problem: Algorithms, large-scale test problems, and computational results. Computers and Operations Research 34(10):2918–2930, DOI 10.1016/j.cor.2005.11.018

Li H, Yao T, Ren M, Rong J, Liu C, Jia L (2016) Physical topology optimization of infrastructure health monitoring sensor network for high-speed rail. Measurement 79:83–93

Lust T, Teghem J (2012) The multiobjective multidimensional knapsack problem: A survey and a new approach. International Transactions in Operational Research 19(4):495–520

Marinakis Y, Marinaki M (2014) A bumble bees mating optimization algorithm for the open vehicle routing problem. Swarm and Evolutionary Computation 15:80–94, DOI 10.1016/j.swevo.2013.12.003, URL http://dx.doi.org/10.1016/j.swevo.2013.12.003

Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. John Wiley & Sons

Martello S, Pisinger D, Vigo D (2000) The three-dimensional bin packing problem. Operations Research 48(2):256–267

Mavrotas G, Florios K, Figueira JR (2015) An improved version of a core based algorithm for the multi-objective multi-dimensional knapsack problem: A computational study and comparison with metaheuristics. Applied Mathematics and Computation 270:25–43

Ortiz-Bayliss JC, Terashima-Marín H, Conant-Pablos SE (2016) Combine and conquer: an evolutionary hyper-heuristic approach for solving constraint satisfaction problems. Artificial Intelligence Review 46(3):327–349

Petursson KB, Runarsson TP (2016) An evolutionary approach to the discovery of hybrid branching rules for mixed integer solvers. Proceedings - 2015 IEEE Symposium Series on Computational Intelligence, SSCI 2015 pp 1436–1443

Pisinger D (2005) Where are the hard knapsack problems? Computers & Operations Research 32(9):2271–2284

Reinelt G (1991) TSPLIBA Traveling Salesman Problem Library. ORSA Journal on Computing 3(4):376–384, DOI 10.1287/ijoc.3.4.376, URL `http://pubsonline.informs.org/doi/abs/10.1287/ijoc.3.4.376`

Samavati M, Essam D, Nehring M, Sarker R (2017) A methodology for the large-scale multi-period precedence-constrained knapsack problem: an application in the mining industry. International Journal of Production Economics 193:12–20

Simon J, Apte A, Regnier E (2017) An application of the multiple knapsack problem: The self-sufficient marine. European Journal of Operational Research 256(3):868–876

Smith-Miles K, van Hemert J (2011) Discovering the suitability of optimisation algorithms by learning from evolved instances. Annals of Mathematics and Artificial Intelligence 61(2):87–104

Smith-Miles K, van Hemert J, Lim X (2010) Understanding tsp difficulty by learning from evolved instances. In: Blum C, Battiti R (eds) Learning and Intelligent Optimization, Lecture Notes in Computer Science, vol 6073, Springer Berlin Heidelberg, pp 266–280

Szkaliczki T, Eberhard M, Hellwagner H, Szobonya L (2014) Piece selection algorithms for layered video streaming in P2P networks. Discrete Applied Mathematics 167:269–279

Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. IEEE Transactions on Evolutionary Computation 3(4):257–271

Zitzler E, Laumanns M, Thiele L (2001) SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems pp 95–100