Nat Comput (2016) 15:181–193
DOI 10.1007/s11047-015-9483-8

Evolution of new algorithms for the binary knapsack problem

Lucas Parada · Carlos Herrera · Mauricio Sepúlveda · Víctor Parada

Published online: 25 January 2015
© Springer Science+Business Media Dordrecht 2015

Abstract Due to its NP-hard nature, it is still difficult to find an optimal solution for instances of the binary knapsack problem as small as 100 variables. In this paper, we developed a three-level hyper-heuristic framework to generate algorithms for the problem. From elementary components and multiple sets of problem instances, algorithms are generated. The best algorithms are selected to go through a second step process, where they are evaluated with problem instances that differ in size and difficulty. The problem instances are generated according to methods that are found in the literature. In all of the larger problem instances, the generated algorithms have less than 1 % error with respect to the optimal solution. Additionally, generated algorithms are efficient, taking on average fractions of a second to find a solution for any instance, with a standard deviation of 1 s. In terms of structure, hyper-heuristic algorithms are compact in size compared with those in the literature, allowing an in-depth analysis of their structure and their presentation to the scientific world.

Keywords Automatic generation of algorithms · Combinatorial optimization · Evolutionary computation · Genetic programming · Hyper-heuristic · Knapsack problem

L. Parada (✉) · C. Herrera
Departamento de Ingeniería Industrial, Universidad de Concepción, Calle Edmundo Larenas 215, Concepción, Chile
e-mail: lucasparada20@gmail.com; lucasparada@udec.cl

C. Herrera
e-mail: cherreral@udec.cl

M. Sepúlveda · V. Parada
Departamento de Ingeniería Informática, Universidad de Santiago de Chile, Av. Ecuador 3659, Santiago, Chile
e-mail: mauricio.sepulveda@usach.cl

V. Parada
e-mail: victor.parada@usach.cl

Abbreviations

BKP	Binary knapsack problem
GP	Genetic programming
GPC++	Genetic programming platform for evolving tree structures of code
IKL	In the knapsack list; data structure used by the evolved algorithms
OKL	Out of knapsack list; data structure used by the evolved algorithms
UC	Uncorrelated instance of the binary knapsack problem
WC	Weakly correlated instance of the binary knapsack problem
SC	Strongly correlated instance of the binary knapsack problem
SS	Subset sum instance of the binary knapsack problem
FC	Fitness cases i.e. problem instances of the binary knapsack problem used to evolve algorithms

1 Introduction

The search for efficient algorithms for combinatorial optimization problems is a central goal of research in the field of optimization. Difficult problems arise from real-world situations in management and information systems. To address such problems in general, some simplifications are made, giving rise to theoretical problems that are difficult to solve computationally because they belong to the NP-complete

class of problems (Garey and Johnson 1979; Fortnow 2009). In addition, each time a new theoretical problem is presented, other real world problems also conform to its formulation. Therefore, such theoretical problems can be considered as fundamental problems. Techniques from integer and dynamic programming have not been sufficient to find the optimal solution with low computational time for many particular problems, and there is growing evidence that specific heuristics and meta-heuristics require low computational time. In turn, the problem of finding the appropriate heuristic for a problem, and more specifically for a subset of problem instances, is a laborious task that requires many computational experiments.

The binary knapsack problem (BKP) is a fundamental problem in combinatorial optimization, and because it belongs to the NP-hard class, the BKP has been extensively studied (Martello and Toth 1990). The BKP can be defined as an integer programming problem according to Eqs. (1–3), where $W > 0$ is the capacity of the knapsack and n is the number of items available. The items have a profit $p_j > 0$ and a weight $w_j > 0, j = 1, 2, \ldots, n$. Furthermore, it is assumed that p_j, w_j and W are integers.

$$Max\ Z = \sum_{j=1}^{n} p_j x_j \tag{1}$$

Subject to:

$$\sum_{j=1}^{n} w_j x_j \leq W \tag{2}$$

$$x_j \in \{0, 1\}, \quad j = 1, 2, \ldots, n \tag{3}$$

Since G. Dantzig devised an approximate greedy algorithm in 1954, various approaches from both the exact and approximate worlds have been developed to address the BKP (Pisinger 2005; Martello and Toth 1990). In recent years, different approaches have been developed (Bienstock 2008; Darehmiraki and Mishmast 2007; Kumar and Singh 2010; Angelelli et al. 2010), including applications in the area of DNA computing (Taghipour et al. 2013; Ye and Zhang 2013). An interesting study on state of the art of the exact methods for the BKP was conducted by Pisinger (2005), revealing that the problem is difficult to solve even for small problem instances.

Evolutionary computation is an increasingly popular heuristic approach to solving difficult combinatorial optimization problems, such as the BKP (Affenzeller et al. 2009). In particular, genetic programming (GP) generates structures in the form of computer programs that automatically create the solution method from some high-level specification of the problem. From elementary instructions, this technique develops code to solve various problems of computer science and creates complex engineering structures such as electrical circuits, antennae and controllers (Koza 1992; Koza et al. 2005). In the field of optimization, Burke et al. (2010) proposed the concept of using hyper-heuristics to solve optimization problems. The idea focuses on searching the space of possible heuristics for a specific optimization problem (Burke et al. 2003, 2010). Then, a highly efficient method is built from elementary components, which may also be known heuristics. However, hyper-heuristics creates complex structures composed of many different heuristics in each stage, and therefore it is difficult to interpret them as algorithms for the problem at hand.

In this paper, through evolutionary computation, algorithms for BKP are generated. We interpret the generated hyper-heuristics as algorithms because they are not a concatenation of heuristics and they have an underlying mode of operation that can be analyzed and extracted for further research. These algorithms are sufficiently robust for different types and sizes of problem instances because they contain the basic operations that, combined with the others, are capable of finding feasible solutions to the problem. Unlike most of the existing approaches (Pisinger 2005), the problem is not solved in a move by move fashion but using general criteria, including several sets of moves. One advantage is that these algorithms can be presented to the scientific community because their compact size, along with their efficient and effective operation that could generate new ideas for algorithms.

The paper is organized as follows. First, the algorithmic generation process is described. Then, the "Modeling the problem" section details the steps for generating algorithms. We describe the evolutionary process by summarizing GP, the data structures that support the algorithms, the elemental components or building blocks of the algorithms and the fitness function of the evolutionary algorithm. The "Results" section presents the results of the work in terms of generating critical variables that affect computer performance and the robustness of algorithms against large problem instances. The top three algorithms are provided, including their pseudocode with a detailed description of their composition and operation. Finally, in the conclusions, we provide a summary of the main characteristics of generating algorithms for the BKP.

2 The process of generating algorithms

Algorithms can be automatically developed by generating hyper-heuristics that can be interpreted as algorithms. Designing hyper-heuristics for combinatorial optimization problems is a process consisting of three layers (Alinia et al. 2012). The innermost layer is the problem to be solved, defined by its mathematical formulation. In the case of the BKP, it corresponds to the integer programming

model (Eqs. 1–3). The middle layer consists of low-level heuristics for determining approximate but feasible solutions to the problem. Finally, the heuristics selection process is at top level of the hyper-heuristic framework. Addressing a different combinatorial problem most likely requires a different set of heuristics, but the approach used to select heuristics is only determined by the experimenter. Moreover, correctly choosing this approach will determine the scope of the study because the act of choosing a heuristic or groups of heuristics is itself a combinatorial problem. For example, Sabar et al. (2012) tested four hyper-heuristics arbitrarily defined by the authors, while Bilgin et al. (2012) tested all possible combinations of a set of initially defined heuristics. Testing only a subset of combinations may seem biased, but trying all possible combinations is a brute force method to address a combinatorial problem. Another option would be to randomly choose the appropriate heuristic for a particular move at each stage of the search process (Demeester et al. 2012).

The heuristic selection process can be performed through adaptive methods or meta-heuristics that search on the solution space. The search process finds heuristics that deliver the best option in every move according to some performance measure (Rajni and Chana 2013; Garrido and Castro 2012; Alinia et al. 2012). Of particular interest, then, is evolutionary computation, the discipline concerned with studying models and tools that use computers to emulate the theory of evolution. Thus, an evolutionary algorithm could search the space of heuristics. The “learning” of the evolutionary algorithm in the heuristic search process generates hyper-heuristics with good results in terms of both solution quality and computational efficiency (Gómez and Terashima-Marín 2012; Burke et al. 2012; Ahmed et al. 2011). The hyper-heuristic approach has been positioning itself as an interesting alternative to address optimization problems; however, the act of choosing a different heuristic at each stage of the search process makes it difficult to generalize the resulting hyper-heuristics to other combinatorial problems.

The current trend with hyper-heuristic is to further isolate each addressed problem by generating a specific method for solving it. Actual hyper-heuristics are a concatenation of many problem-specific heuristics (Gómez and Terashima-Marín 2012; Burke et al. 2012; Ahmed et al. 2011). Then, the solution method can be thought of one process consisting of many moves, where each move is performed by a different heuristic which in turn, represents different criteria for every move. If there is a unifying theory of classical combinatorial optimization problems, the search methods that particularize each problem solution only serve to move in the opposite direction of that theory. Additionally, such hyper-heuristics made by a concatenation of many individual moves provide a small contribution to obtaining a robust algorithm for the problem. Consider the extreme situation where the framework determines that every move will be made by a different heuristic. Then, for a large instance of the problem, the hyper-heuristic will have as many heuristics as moves needed to solve the problem. The resulting solution method will look like a large formula with no apparent structure behind it, other than being greedy in every move. Few new algorithmic ideas can be extracted from such a method.

Using high-level instructions and existing heuristics for the BKP, hyper-heuristics interpretable as new algorithms can be produced by GP. The goal of this technique is to automatically design computer code for a problem given high-level definitions. Populations of individuals ranked according to their quality, evolve, guided by some evaluation function defined by the experimenters using the GP operators. Individuals are syntax trees that represent computer programs. The internal nodes of a tree are algorithmic functions, while the leaf nodes correspond to actions that are performed on the data structure containing a solution for the BKP. The best individuals are selected, crossed over and mutated (mutation could randomly be shrink or subtree). Finally, to guide the search process, a fitness function is used, which may include different criteria associated with the product to obtain. When automatically generating algorithms three key criteria should be considered: the error of the algorithm when solving a set of instances, a measure of its effectiveness in terms of the number of times an optimal solution is found and its size measured in number of nodes (Koza et al. 2005; Poli et al. 2008). The pseudocode of GP is given next:

```
01  Initialize(Population(0));
02    generation = 0;
03    While(Stopping criteria == FALSE) do
04        Evaluate(Population(generation));
05        Parents = Select(Population(generation));
06        Children = ApplyEvolutiveOperators(Parents);
07        NewPopulation = Replace(Children,Population(generation));
08        generation++;
09        Population(generation) = NewPopulation
10    End While
11  Return BestSolutionFound;
```

This article aims to generate hyper-heuristics decodable as algorithms by means of GP. We named this process the automatic generation of algorithms. In this article, algorithms for the BKP are automatically generated. In evolving these algorithms, GP is used and is charged with selecting adequate functions and terminals for the BKP. Figure 1 shows this process as a structured analysis and design diagram. Specifically, Fig. 1a) shows the general process in which the analyst defines the data structure (DS) and the set of terminals (T) from their knowledge. Then, it is integrated into the automatic process. Figure 1b) shows the automated process implemented by the GP platform. In this way, a new algorithm is generated after each

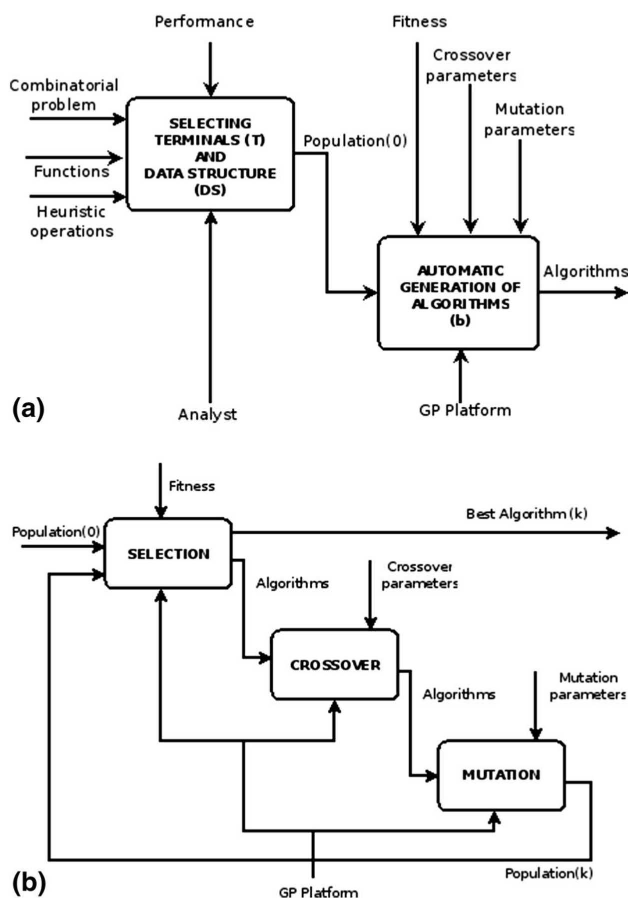

Fig. 1 The process to automatically generate algorithms

generation, and the process ends after k generations. In this paper, we describe three highly efficient and robust algorithms for the BKP generated by this methodology.

3 Modeling the problem

3.1 Genetic programing and the evolutionary process

This section describes the process for obtaining evolutionary algorithms. The data structures are defined, and the fitness function is presented along with the corresponding sets of functions and terminals, which includes basic low-level heuristics, that give rise to algorithms. Likewise, the hardware and software features used are described.

The GP framework applied in our experiment derives from the following processes. First, to evolve the algorithms, a set of functions and terminals must be created. Those two sets must contain the elemental components that the algorithms will possess. When provided those definitions, an initial population of syntax trees can be configured and successively evolved in a sequence of future generations using the evolutionary operators. From generation to generation, individuals are selected according to training problem instances (fitness cases) and a fitness function specifically designed for the BKP. The syntax trees are decoded in their corresponding algorithms (in pseudocode) and are evaluated externally with evaluation sets of problem instances (evaluation cases). Thus, the construction of algorithms considers two stages: first, the evolution occurs with sets of problem instances, with an emphasis on improving the fitness of each algorithm. In the second stage, the generated algorithms are evaluated with evaluation cases.

To perform the evolutionary process, a computational program that evolves syntax trees is used. The trees are sets of instructions that can be executed following a predetermined order. This task is accomplished with the platform GPC++ (Fraser and Weinbrenner 1993). In order to determine the most appropriate population size for generating algorithms, the following values were tested: 1,000, 2,000, 3,000, 4,000 and 5,000 individuals. To generate the initial population, the ramped half and half method was used, with crossover probabilities of 0.85 and 0.05 for the mutation (0.03 and 0.02 for shrink and sub tree mutation, respectively). Furthermore, it was established that 300 generations of the evolutionary algorithm would be the stopping criterion. To ensure robustness to the randomness inherent in the process, each execution of GPC++ was repeated 30 times with different random seeds.

3.2 Data structures of the algorithms

The algorithms are supported on four data structures. The IKL (IN THE KNAPSACK LIST) is defined as a list of elements that are in the knapsack at a given time, while the OKL (OUT OF KNAPSACK LIST) is the complement of IKL, i.e., the list containing all potential elements for inclusion in the knapsack. The third and fourth data structures were defined according to theoretical BKP results (Balas and Zemel 1980). We define a list called the CORE LIST whose elements consist of the neighboring items to the last one that fits in the backpack if these were inserted one at a time, in decreasing order by the ratio p_j/w_j. The CORE LIST elements can be inserted or removed from the knapsack according to the criterion of the generated algorithms. Similarly, we define the WEIGHT LIST, whose elements consist of the neighboring items to the last one that fits in the knapsack, if these were inserted one by one, in decreasing order by their weight w_j.

3.3 Function and terminals definition

The functions and terminals are the basic elements used to influence the defined data structures (Koza et al. 1997; Poli et al. 2008). The terminals are the leaf nodes of the tree and

allow changes to the data structures of the problem. In our case, the terminals are functions specially designed for the BKP that select an element and insert or remove it from the knapsack, according to some predefined criteria. Specifically, five insertion terminals and three deletion ones are defined, each with a single criterion. The five insertions have the following criteria for any item: maximum weight, minimum weight, maximum profit, maximum profit/weight ratio and being the first to fit in the knapsack. Along the same lines, the three criteria for deleting items are the following: maximum weight, minimum profit and profit/weight. Each terminal checks the feasibility of its operation, i.e., a terminal operates if and only if the move is feasible for the BKP. We define two additional terminals that act as flags, i.e., no immediate action, but when called upon, each flag indicates that the next operation on the data structures must be performed on only a specific subset of elements. Then, the terminal *Set_Core* indicates that the next operation (insertion or deletion) must be performed on an element of the CORE LIST. The *Set_Weight* terminal operation is analogous to the *Set_Core*, but the former operates with the WEIGHT LIST. Finally, each terminal was designed to return an integer value to internally keep track of the created algorithm. This integer value is also taken as a logical variable. We adopted the convention that a value of zero is false and a value greater than zero is true. Thus, the insertion and deletion terminals return the value that corresponds to the operated item number (true) or zero in case no item was selected (false). The two terminals designed as flags always return the value one (true).

The functions that shape the algorithms are control flow statements typically found in most programming languages. They are *While*, *And*, *Or*, *If_Then*, *Not* and *Equal*. As with the terminals, the functions were designed to return integers, again using the convention that a value of zero represents false, while any other value (integer greater than zero) represents true. Thus, for P1 and P2, which may be functions or terminals, there is *While*(P1,P2), which executes the instruction P2 and returns the number of executions while P1 is true. This role is key in the generation of algorithms because it allows grouping sets of instructions. For the definition of the "While" and generally for all functions presented here, P1 is not merely a typical stopping criterion because it can now become a complex structure integrated by several terminal and/or functions. Likewise, P2 could also be a complex set of instructions. This new definition of a "While" loop and other functions can generate generalizable algorithms from different combinatorial optimization problems. Then, *If_Then*(P1,P2) executes P2 and returns its integer value if P1 is true, while *Not*(P1) executes P1 and returns the logical complement of the variable associated with the integer value of P1. The function *Equal*(P1,P2) returns true if P1 and P2 return the same logical variables. Finally, *And*(P1,P2) returns true if and only if P1 and P2 are true.

Each insertion or deletion terminal only searches through the items for the largest or smallest value of a given criteria. There is no need to sort them in the process, although an argument can be made to previously sort the items according to each different criterion and measuring the efficiency of doing so. Then, the mixture of the terminals with the different functions achieves a great deal of work. For example, if only two nodes are needed to fill a knapsack: *While*(P1,P2) plus any insertion terminal will produce a full knapsack. Despite this, during preliminary studies we found that without any control over the resulting tree size, the algorithms could grow to hold thousands of nodes. The maximum tree height was set at 14 levels in depth, which is equivalent to having a tree with at most 32,767 nodes. Thus, stronger control over tree size was necessary, as any algorithm with thousands of instructions was impractical for our study. We set 14 nodes as the target algorithm size because the studies showed that the evolutionary process could generate good quality individuals with that many nodes.

The logic behind the creation of the terminals and functions consists of decomposing existing heuristics for BKP into a set of basic moves over the data structure. Thus, we can express the problem of finding an algorithm as finding the set of correct moves that produces the highest possible value of a number of performance measures expressed as a fitness function. For example, in the BKP, one basic move can be inserting one item into the knapsack according to a given criteria, say the one with the highest profit. Another basic move may consist of removing a specific item from the knapsack according to another given criteria, for example, the item with the highest weight. The problem here lies in the fact that there may be an infinite set of potential basic moves. Our terminals and functions then attempt to develop a very small set of basic moves for the BKP that are an effort to represent the inherent combinatorial nature of the problem.

3.4 The fitness function

The fitness function is responsible for guiding the search process to find new algorithms and considers three criteria. Those are (i) the error generated by a new algorithm to solve a set of BKP instances regarding the optimal solution (ii) the number of obtained solutions that happen to be optimal (also named hits) and (iii) the algorithm size measured by the number of nodes. Thus z_j is the value obtained when solving for instance j, and u_j is the optimal value of the objective function for this instance. Furthermore, n_f is the number of problem instances, h is an integer counter of the number of hits (optimal solutions or when

the solution is 0.001 % or less away from the optimal). l_a the number of nodes of an algorithm, and α, β are parameters that establish the importance of each term in the fitness function. Moreover, let t_a be the desired size for the resulting algorithm measured by the number of nodes. Then, a fitness f_a algorithm is given by Eq. 4.

$$f_a = \frac{\propto}{n_f}\sum_{j=1}^{n_f}\frac{|u_j - z_j|}{z_j} + \beta(1-\propto)\left(1-\frac{h}{n_f}\right) + \beta(1-\propto)\left(\frac{|t_a - l_a|}{t_a}\right) \quad (4)$$

4 Numerical results

4.1 Computational time required to evolve algorithms

Two critical factors that influence the computational time when experimenting with the automatic generation of algorithms are the size of the population and the size of the individuals. Because evaluating a candidate heuristic requires executing it on several test BKP instances, this step is slow. Similarly because evaluations of an individual consist of executing each instruction for a number of different problem instances, the greater the number of nodes in a tree, the greater is the computational time required.

A preliminary study was performed that found that the population size did not affect the resulting quality of the algorithms, as measured by the number of optimal solutions or hits. For each defined population size, GPC++ was executed 30 times using eight training instance with a stopping criterion of 300 generations, obtaining a total of 150 algorithms. The primary difference was in the computational time involved, as shown in Fig. 2. We detect significant variability of times for different runs, which is observed even among the 30 executions of the same population size. The times seem to have, at first glance, no relation to the population size because, for example, there are points (runs) for the population sizes of 2,000 and 3,000 with duration of more than 250,000 s, which is equivalent to almost 4 days on the computer.

It is expected that the larger the population size, the higher the duration of executions due to a greater number of evaluations, selections and crossovers for each generation. However, in Fig. 2 it is observed that even within the same population size, no relationship between the durations of the runs appears. Observe, for example, the population size of 1,000, whose points appear to be close together. However, the output files of the platform show that the fastest execution was 40 min, while the longest was more than 10 h. Thus, in relation to the time of evolution, within each population size, the execution times can be very short

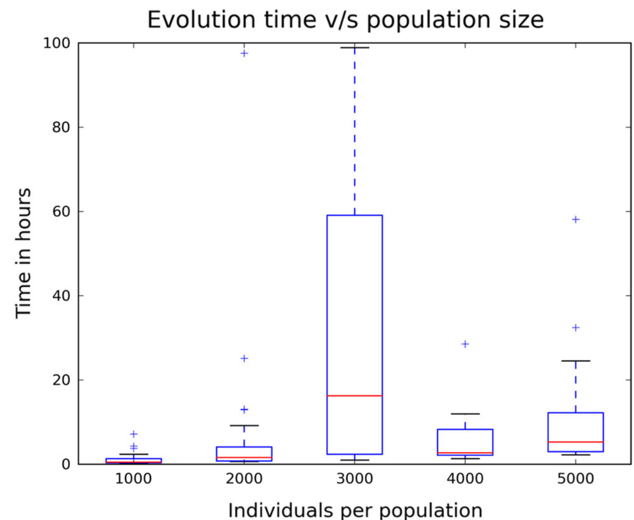

Fig. 2 Computational time involved in the determination of an arbitrary population size

or very long. A hypothesis was defined that stated that all population sizes had the same average running time. This hypothesis was rejected by ANOVA, indicating that there are population sizes where the computational times are indeed smaller on average. Consequently, we decided to use the smaller size of 1,000 individuals per population.

A study was conducted to analyze the relationship between the size of an algorithm, given by its number of nodes, and its quality. It emerged that, without some control over the size, i.e. regardless of the term corresponding to the number of nodes in Eq. 1, the resulting algorithms can contain thousands of nodes; however, this number had no great impact on quality. Thus, 14 nodes were determined to be the desired number of nodes for an algorithm.

4.2 Robustness of the generated algorithms

One factor that affects the runtime of automatically generating algorithms is the number of problem instances used as fitness cases. Similar to the approach proposed to address the size of the population, each population algorithm must resolve all fitness cases through all generations. Naturally, with a large number of fitness cases, greater computational effort is necessary. In addition, it is important that the generated algorithms maintain their performance when processing the different fitness cases. This study determines an arbitrary number of fitness cases, and the robustness of algorithms is tested against large problem instances of the BKP. To this end, two groups of problem instances were obtained: 24 problem instances for evolution and 140 to evaluate the generated algorithms.

All problem instances were produced using a random generator developed by Pisinger (2005). The generator is

capable of producing different problem instances according to different values of the p_j/w_j ratio. Depending on these values, the problem instances can be classified into four types: Uncorrelated (UC), Weakly Correlated (WC), Strongly Correlated (SC) and Subset Sum (SS). Therefore, 24 fitness cases were generated, with 6 problem instances of each type and 100 items in the knapsack for each. Fitness cases were divided into five groups: FC_8, FC_{12}, FC_{16}, FC_{20} and FC_{24} with 8, 12, 16, 20 and 24 problem instances, respectively. For example, FC_8 consists of two problem instances of each type, FC_{12} consists of three problem instances of each type and so on. For each problem instance, a defined upper bound was calculated by Martello and Toth (1990). Because each execution of GPC++ was repeated 30 times, the evolution with a set of fitness cases generated 30 algorithms (the top 30 of each process) and a total of 150 best algorithms for the five fitness cases. The 140 problem instances produced for the evaluation of algorithms were divided into seven groups of 20 problem instances called S_{100}, S_{200}, S_{500}, $S_{1,000}$, $S_{2,000}$, $S_{5,000}$ and $S_{10,000}$, where the sub index value indicates the number of available items that can go in the knapsack. Each of these sets contains five problem instances of the four groups: five UC, five WC, five SC and five SS.

The produced algorithms significantly improved their performance when evaluated with large problem instances. However, there was no improvement in quality when using more fitness cases for training, as shown in Tables 1 and 2. Specifically, Table 1 lists the number of "hits" obtained by the best algorithms. A hit is defined as a variation of less than 0.001 % between the solution delivered by a generated algorithm and the best solution available for the problem. Then, a maximum of 600 hits can occur in each cell, the product of the 30 algorithms with 20 cases of evaluation. The number of hits increases as one moves to the right of the table, which indicates the robustness of the algorithms for larger problem instances. This situation occurs because an increase in the number of items involves a greater number of favorable combinations that can generate a higher value in the objective function. Furthermore, when analyzing the columns of Table 1, there is no improvement in quality when using more fitness cases. In any column, the number of hits has little variation in relation to the total, a fact that is expressed by the low standard deviation values.

Table 1 Hits of the best 150 algorithms

	S_{100}	S_{200}	S_{500}	$S_{1,000}$	$S_{2,000}$	$S_{5,000}$	$S_{10,000}$
FC_8	60	172	143	226	226	293	310
FC_{12}	56	161	141	216	223	293	309
FC_{16}	58	176	148	235	239	302	313
FC_{20}	54	165	142	224	228	305	320
FC_{24}	58	169	146	231	238	297	314
Average	57.2	168.6	144.0	226.4	230.8	298.0	313.2
SD	2.3	5.9	2.9	7.2	7.3	5.4	4.3

The largest numbers of hits were obtained when evaluating larger problem instances (10,000 items). In particular, we highlight the 30 best algorithms obtained from 20 fitness cases whose hits count reached 320 according to Table 1. Table 2 presents the details of these algorithms, where it can be observed that all algorithms are of high quality and provided low relative errors with respect to the optimal solution. Table 2 is read as follows: the columns represent the four groups of problem instances, where each group consists of five problem instances. The rows have the 30 best algorithms generated using 30 different seeds. They are called KPA, as in "Knapsack Algorithm". For each seed we obtain values that corresponding to the best (nearest to the optimal) and the worst (furthest from the optimal) solution and the average and standard deviation of the relative error with respect to the optimal solution for each group of five problem instances. For simplicity, the relative error is shown as the difference measured in percentages. In addition, for each group, two columns are included that analyze the existence of an optimal solution. The column "hits" indicates, as in Table 1, that the solution in one of the five cases is at most 0.001 % away from the available optimal solution. The column "aggregated hits" sums all of the hits of the 30 evaluated algorithms. Of particular interest is the last cell of this last column because it shows the total number of optimal solutions within a group, and thus, the sum of these four cells (one per group) shows the total optimal solutions found by the 30 algorithms. The relative errors for this group of algorithms and problem instances as a whole are the lowest of the entire study, with fractional values of the order of hundredths in all cases.

4.3 Three of the best generated algorithms

Although many of the algorithms found have similar characteristics and are of the same size and have a low error in their fitness, in this section, we describe the three best algorithms found, named KPA1, KPA5 and KPA13. To clearly understand their functioning they are also presented in pseudocode. The algorithm becomes more readable once it is put in pseudocode. Readability is not only a visual aid but contributes to understanding the solution process followed by an algorithm. The difference compared to a tree representation of instructions is that in the pseudocode, the functions that operate only as connectors between terminals can be dispensed, as occurs with functions *If_then*, *Not* and *Equal* functions.

Table 2 Numerical results of algorithms evolved with 20 fitness cases and evaluated with $S_{10,000}$

20/10000 items	SC instances						UC instances					
Id.	Average %	Worst %	Best %	SD	Hits	Agg. Hits	Average %	Worst %	Best %	SD	Hits	Agg. Hits
KPA1	0.00	0.00	0.00	0.00	5	5	0.01	0.02	0.00	0.00	0	0
KPA2	0.00	0.00	0.00	0.00	5	10	0.00	0.01	0.00	0.00	0	0
KPA3	0.00	0.00	0.00	0.00	5	15	0.00	0.01	0.00	0.00	0	0
KPA4	0.00	0.00	0.00	0.00	5	20	0.00	0.01	0.00	0.00	0	0
KPA5	0.00	0.00	0.00	0.00	5	25	0.00	0.01	0.00	0.00	0	0
KPA6	0.00	0.00	0.00	0.00	5	30	0.01	0.02	0.00	0.00	0	0
KPA7	0.00	0.00	0.00	0.00	5	35	0.01	0.02	0.00	0.00	0	0
KPA8	0.00	0.00	0.00	0.00	5	40	0.00	0.01	0.00	0.00	0	0
KPA9	0.00	0.00	0.00	0.00	5	45	0.00	0.01	0.00	0.00	0	0
KPA10	0.23	0.35	0.16	0.07	0	45	0.00	0.01	0.00	0.00	0	0
KPA11	0.00	0.00	0.00	0.00	5	50	0.00	0.01	0.00	0.00	0	0
KPA12	0.23	0.35	0.16	0.07	0	50	0.00	0.01	0.00	0.00	0	0
KPA13	0.00	0.00	0.00	0.00	5	55	0.00	0.01	0.00	0.00	0	0
KPA14	0.23	0.35	0.16	0.07	0	55	0.00	0.01	0.00	0.00	0	0
KPA15	0.23	0.35	0.16	0.07	0	55	0.00	0.01	0.00	0.00	0	0
KPA16	0.00	0.00	0.00	0.00	5	60	0.00	0.01	0.00	0.00	0	0
KPA17	0.00	0.00	0.00	0.00	5	65	0.00	0.01	0.00	0.00	0	0
KPA18	0.00	0.00	0.00	0.00	5	70	0.00	0.01	0.00	0.00	0	0
KPA19	0.00	0.00	0.00	0.00	5	75	0.00	0.01	0.00	0.00	0	0
KPA20	0.00	0.00	0.00	0.00	5	80	0.01	0.02	0.00	0.00	0	0
KPA21	0.23	0.35	0.16	0.07	0	80	0.00	0.01	0.00	0.00	0	0
KPA22	0.00	0.00	0.00	0.00	5	85	0.01	0.02	0.00	0.00	0	0
KPA23	0.00	0.00	0.00	0.00	5	90	0.01	0.02	0.00	0.00	0	0
KPA24	0.23	0.35	0.16	0.07	0	90	0.00	0.01	0.00	0.00	0	0
KPA25	0.06	0.12	0.03	0.04	0	90	0.00	0.01	0.00	0.00	1	1
KPA26	0.00	0.00	0.00	0.00	5	95	0.01	0.02	0.00	0.00	0	1
KPA27	0.00	0.00	0.00	0.00	5	100	0.01	0.02	0.00	0.00	0	1
KPA28	0.00	0.00	0.00	0.00	5	105	0.01	0.02	0.00	0.00	0	1
KPA29	0.13	0.18	0.07	0.05	0	105	0.02	0.06	0.00	0.03	0	1
KPA30	0.00	0.00	0.00	0.00	5	110	0.00	0.01	0.00	0.00	0	1
20/10000 items	WC instances						SS instances					
Id.	Average %	Worst %	Best %	SD	Hits	Agg. Hits	Average %	Worst %	Best %	SD	Hits	Agg. Hits
KPA1	0.02	0.07	0.00	0.03	2	2	0.00	0.00	0.00	0.00	5	5
KPA2	0.02	0.07	0.00	0.03	2	4	0.00	0.00	0.00	0.00	5	10
KPA3	0.02	0.07	0.00	0.03	2	6	0.00	0.00	0.00	0.00	5	15
KPA4	0.02	0.07	0.00	0.03	2	8	0.00	0.00	0.00	0.00	5	20
KPA5	0.02	0.07	0.00	0.03	2	10	0.00	0.00	0.00	0.00	5	25
KPA6	0.02	0.07	0.00	0.03	2	12	0.00	0.00	0.00	0.00	5	30
KPA7	0.02	0.07	0.00	0.03	2	14	0.00	0.00	0.00	0.00	5	35
KPA8	0.02	0.07	0.00	0.03	2	16	0.00	0.00	0.00	0.00	5	40
KPA9	0.02	0.07	0.00	0.03	2	18	0.00	0.00	0.00	0.00	5	45
KPA10	0.02	0.07	0.00	0.03	2	20	0.02	0.11	0.00	0.05	4	49
KPA11	0.02	0.07	0.00	0.03	2	22	0.00	0.00	0.00	0.00	5	54
KPA12	0.02	0.07	0.00	0.03	2	24	0.00	0.00	0.00	0.00	5	59
KPA13	0.02	0.07	0.00	0.03	2	26	0.00	0.00	0.00	0.00	5	64

Table 2 continued

20/10000 items	WC instances						SS instances					
Id.	Average %	Worst %	Best %	SD	Hits	Agg. Hits	Average %	Worst %	Best %	SD	Hits	Agg. Hits
KPA14	0.02	0.07	0.00	0.03	2	28	0.00	0.00	0.00	0.00	5	69
KPA15	0.02	0.07	0.00	0.03	2	30	0.00	0.00	0.00	0.00	5	74
KPA16	0.02	0.07	0.00	0.03	2	32	0.00	0.00	0.00	0.00	5	79
KPA17	0.02	0.07	0.00	0.03	2	34	0.00	0.00	0.00	0.00	5	84
KPA18	0.02	0.07	0.00	0.03	2	36	0.00	0.00	0.00	0.00	5	89
KPA19	0.02	0.07	0.00	0.03	2	38	0.00	0.00	0.00	0.00	5	94
KPA20	0.02	0.07	0.00	0.03	2	40	0.00	0.00	0.00	0.00	5	99
KPA21	0.02	0.07	0.00	0.03	2	42	0.00	0.00	0.00	0.00	5	104
KPA22	0.02	0.07	0.00	0.03	2	44	0.00	0.00	0.00	0.00	5	109
KPA23	0.02	0.07	0.00	0.03	2	46	0.00	0.00	0.00	0.00	5	114
KPA24	0.02	0.07	0.00	0.03	2	48	0.00	0.00	0.00	0.00	5	119
KPA25	0.02	0.07	0.00	0.03	2	50	0.00	0.00	0.00	0.00	5	124
KPA26	0.02	0.07	0.00	0.03	2	52	0.00	0.00	0.00	0.00	5	129
KPA27	0.02	0.07	0.00	0.03	2	54	0.00	0.00	0.00	0.00	5	134
KPA28	0.02	0.07	0.00	0.03	2	56	0.00	0.00	0.00	0.00	5	139
KPA29	0.02	0.10	0.00	0.04	2	58	0.00	0.00	0.00	0.00	5	144
KPA30	0.02	0.07	0.00	0.03	2	60	0.00	0.00	0.00	0.00	5	149

Although the three algorithms presented in this paper are similar in terms of quality, KPA1 is the most complex in its mode of operation due to a number of nested *While* loops (Fig. 3). In fact, this algorithm has a single *While* loop whose stopping condition is both branches 1 and 2, while branch 3 represents the block of instructions of the loop. Remarkably, during the automatic generation of algorithms for the BKP, any stopping criteria can also be a set of instructions to be executed. In detail, in branch 1, the knapsack is filled according to the p_j/w_j value of an item (two *Add_Dantzig* terminals). As for algorithm 2 (KPA5), branches 2 and 3 of this KPA1 also operate as a refinement of the solution. First, an exchange is made between the item with the worst profit/weight ratio and the one with the highest profit that is not in the knapsack. The branch ends with the terminal *Set_Core*, indicating that the first operation of branch 3 must be run from the CORE LIST. Finally, in branch 3, an exchange is made between the item in the knapsack that has the worst profit and the element outside of the knapsack that has the most profit. KPA1 starts with three hits with evaluation in cases of 100 items and reaches 12 hits in the case of evaluation with 10,000 items (See Table 3). This algorithm obtained a total of 64 hits in the evaluation cases. The pseudocode of KPA1 is given below.

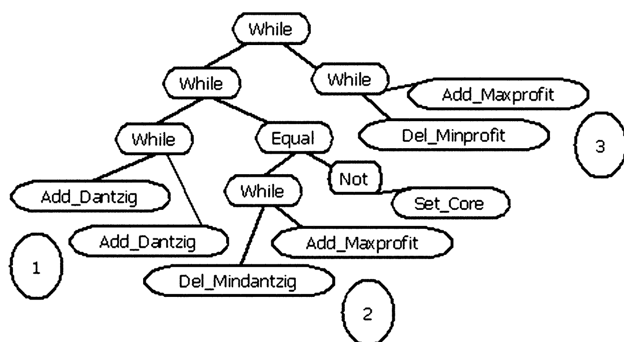

Fig. 3 Algorithm KPA1 evolved from 16 fitness cases

Algorithm KPA1 from FC_{16}

```
01  DEFINE Inst_size    // the size of the instance is defined as an integer equal
    to the number of items in the given instance.
02  n = 1
03  While ( KP != FULL && KP != EMPTY && n <= Inst_size)
04          i = 1;
05          While ( KP != FULL && KP != EMPTY && i <= Inst_size)
06                  While ( KP != FULL) Add_Dantzig;
07                  j = 1;
08                  While ( KP != EMPTY && j <= Inst_size)
09                          Del_Mindantzig;
10                          Add_Maxprofit;
11                          j++;
12                  End While
13                  Set_Core;
14                  k = 1;
15                  While ( KP != EMPTY && n <= Inst_size)
16                          Del_Minprofit;
17                          Add_Maxprofit;
18                          k++;
19                  End While
20          i++;
21          End While
22  n++;
23  End While
```

Table 3 Hits of the best three algorithms compared to their peers

	S_{100}	S_{200}	S_{500}	$S_{1,000}$	$S_{2,000}$	$S_{5,000}$	$S_{10,000}$
KPA13	4	9	7	12	10	12	12
KPA5	3	7	7	12	11	12	12
KPA1	3	7	7	12	11	12	12
Average	1.90	6.52	4.76	7.51	7.53	9.80	10.33
SD	0.75	1.23	0.81	2.53	1.91	2.22	2.45

The pseudocode of KPA1 shows how versatile the automatic generation of algorithms for the BKP is. The structure of this algorithm is more complex than the other two, but its effectiveness in resolving problem instances is as high as the other two. The highlights of KPA1 are the *While* loops in lines 03 and 05 because both have three possible stopping conditions. Unless the type of problem instance to be solved is known, which implies an approximate knowledge about the profit/weight ratio of the items in the BKP, it is not possible to anticipate how the first two loops in KPA1 will end, which indicates that KPA1 operates differently for different types of problem instances. Algorithms adaptable to different types of problem instances are robust and generalizable to other instances. These features are highly desirable and are precisely what is sought when automatically generating a solution method.

With a total of 64 hits considering all evaluation cases from the 12 fitness training cases (FC_{12}), KPA5 is the second-best algorithm obtained according to the criterion of the number of hits. The structure of this second algorithm is presented in Fig. 4. The algorithm is divided into three branches to facilitate understanding. In branch 1, the knapsack is filled. Before this filling, KPA5 invokes the terminal *Set_Weight*, which states that the first element must be obtained from the WEIGHT LIST. Then, it goes to the *While* loop that iterates until the terminal *T_True* returns false. Thus, the cycle will continue until it meets one of the following two conditions: no more elements can be inserted into the knapsack (*Add_Dantzig* terminal condition) or it calls a terminal *Add_Dantzig n* times, where *n* equals the total number of possible items to be inserted into the knapsack. Trivially, the first condition met will be the capacity of the knapsack, leaving branch 1 with a knapsack filled according to the criterion of item efficiency. To continue to operate on the various data structures of the knapsack, an item should be removed to thus "make room" for potential items that were not considered and that could lead to a better solution.

KPA5 can be viewed as a refined Dantzig's algorithm because branch 1 corresponds with this algorithm while branches 2 and 3 refine and improve the search process through the insertion and removal of elements according to different criteria. In branch 2, removing items is performed through the terminal *Del_Maxweight*, i.e., the item with the largest weight is removed from the knapsack to make room to insert the item with the greatest profit (*Add_Maxprofit*). Finally, branch 3 has a refining process; it exchanges the item with the worst profit/weight for the one with the most weight. In terms of hits, KPA5 finds five optimal results when evaluating100 items from the knapsack; that number increases to 12 hits with problem instances of 10000 items (See Table 3). Overall, branch 2 has more influence on the final solution than branch 3 because branch 3 only performs one exchange of items of the BKP. Furthermore, a priori one cannot say if this exchange is for better or worse, i.e., the exchange in branch 3 only improves the solution if the item being inserted has a higher profit than the item coming out of the knapsack, which in turn depends on the type of problem instance.

Similar to KPA1, being able to dispense of the functions *Equal*, *Not* and *Or* increases the readability of the algorithm KPA5, as in the pseudocode given below.

Algorithm KPA5 from FC_{12}

```
01 DEFINE Inst_size     // the size of the instance is defined as an integer equal to
   the                              number of items in the given instance.
02 Set_Weight;
03 While ( KP != FULL) Add_Dantzig;
04 n = 1;
05 While ( KP != EMPTY && n <= Inst_size)
06         Del_Maxweight;
07         Add_Maxprofit;
08         n++;
09 End While
10 Del_Mindantzig;
11 Add_Maxweight;
```

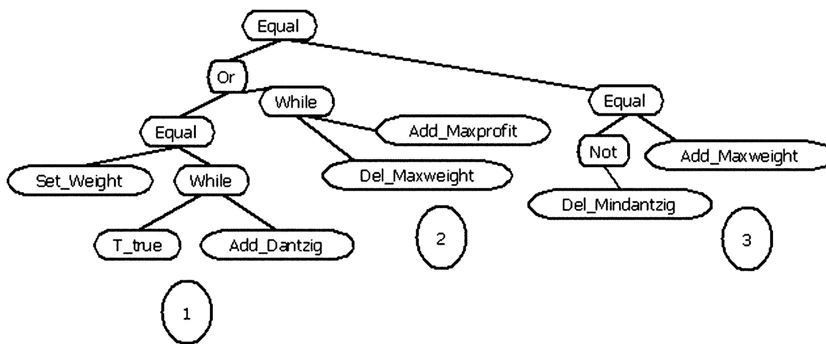

Fig. 4 Algorithm KPA5 evolved from 12 fitness cases

With 66 hits from eight training problem instances (FC_8), KPA13 is the best algorithm found (Fig. 5). Its method of operation is relatively trivial; KPA13 is divided into three branches distinguished by a circled number next to the branch. First, in branch 1, it inserts the first item into the knapsack according to the minimum weight criteria. This item is obtained from the CORE LIST because the terminal *Set_Core* was called. Then, in branch 2, it proceeds to fill the knapsack through a *While* loop whose insertion terminal is *Add_Dantzig*. In other words, the knapsack is filled according to the profit/weight ratio of each item in decreasing order. Finally, in branch 3, an “adjustment” is made to the current solution, eliminating those items with the worst profit/weight ratio (*Del_Mindantzig*) and inserting those with the highest weight (*Add_Maxweight*). The mode of operation of KPA13 can be observed as a three-stage process: pretreating the data, filling the knapsack and refining the solution. These stages can be found in most of the best algorithms generated. Figure 5 shows that there are algorithmic structures and sub nodes that are redundant, in the sense that they do not alter the various data structures. For example, in branch 1, there are two terminals that trigger the CORE LIST, but only one of them is sufficient. Unless they are succeeded by a terminal of insertion or deletion, *Set_Core* has no effect. Another example is the *Not* function, also in branch 1. Because it is in a nested *Equal* that in turn belongs to an *And*, it does not matter whether the structure *Not*(*Add_Minweight*) is true or false. The pseudocode of KPA13 is given below.

To write KPA13 in pseudocode, first we define an integer corresponding to the size of the problem instance. This number will act as a stopping criterion for the defined *While* cycles. However, this condition is only valid in the cycle of line 06. In line 04, the loop will only end due to knapsack capacity because BKP instances have a much larger number of items than can fit in the knapsack. For the cycle in line 06, both conditions are required because we cannot anticipate which will be met first; it depends on the problem at hand.

Algorithm KPA13 from FC_8

```
01    DEFINE Inst_size      // the size of the instance is defined as an integer equal
to the                                    number of items in the given instance.
02    Set_Core;
03    Add_Minweight;
04    While ( KP != FULL) Add_Dantzig;
05    n = 1;
06    While ( KP != EMPTY && n <= Inst_size)
07              Del_Mindantzig;
08              Add_Maxweight;
09              n++;
10    End While
```

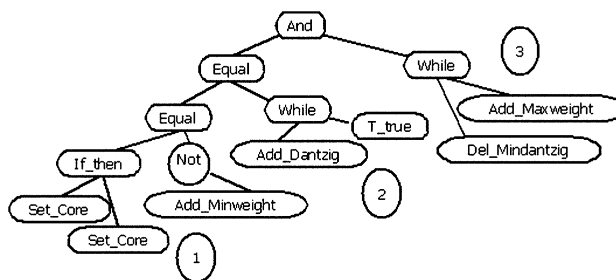

Fig. 5 Algorithm KPA13 evolved from 8 fitness cases

Because the total number of evaluation cases is 140, KPA5 has a difference of merely two hits fewer than KPA13. This suggests that the two have virtually the same quality. By analyzing both pseudocodes with regard to the similarities, there are two *While* loops, and their underlying objectives are as follows: First, fill the knapsack and then refine the solution. The differences are the various terminals that that the algorithms use. For example, the KPA13 refinement is performed with the instructions *Del_Mindantzig—Add_Maxweight*, while KPA5 employs *Del_Maxweight—Add_Maxprofit* to perform the same task.

To analyze how “good” these three algorithms are in relation to their peers, Table 3 is presented. Table 3 has the number of hits obtained by the previous three algorithms in its first three rows, shown with different cases of evaluation from S_{100} to $S_{10,000}$. The fourth and fifth rows reflect data from the rest of the 147 best algorithms obtained in the study. The average row indicates the average number of hits obtained by the rest of the algorithms, while the SD row is the standard deviation of the recorded average values, fulfilling the function of a complementary measure of dispersion. It is observed that the three best algorithms outperform the average for each case of adaptation. Furthermore, each of these three best algorithms exceeds the total of the average by over 10 hits (adding each row to the right). Because the average of the algorithms reaches a total of 51 hits (rounded up to the nearest integer), the difference of more than 10 hits means that each of the three best algorithms is at least 20 % better than average.

The 150 algorithms generated during the evolutionary process solved the 140 problem instances with low computational time. The time required for all 21,000 evaluation executions (150 times 140) was 1 h and 58 min (approximately 7,132 s). Thus, for each evaluation, on average, 0.34 s were required per instance and algorithm, with a standard deviation of 1 s. The highest times were 10 min when solving some 10,000 item problem instances but for only a limited number of algorithms. For the vast majority, however, the time required was only fractions of a millisecond.

5 Discussion

Of the three algorithms presented, there are important perspectives to highlight. For example, inserting elements into the knapsack is always carried out according to the profit/weight ratio in descending order using our *Add_Dantzig* terminal, which indicates that of the five insertion terminals, *Add_Dantzig* was always privileged by evolution over its peers. This aspect is not minor for the automatic generation of algorithms for the BKP because the algorithms can also refine the solution in addition to inserting any item. While the refining process sometimes totally changes the initial solution generated by the initial insertion of elements, it is preferable that the items be inserted appropriately. In other words, the initial order or arrangement of the elements in the knapsack does matter for generating quality solutions. Therefore, the structure *While(Add_Dantzig)* can be regarded as a successful structure for GP because it is present in the best algorithms. Another result of analysis comes from the exploitation of the terminals used. Given that the best algorithms have repeated terminals that do not contribute to the solution of the BKP, the imposed number of 14 terminals is still high. Algorithms can obtain similar quality with fewer nodes, facilitating monitoring the operation of the algorithms.

When analyzing the pseudocodes shown, the algorithms could operate differently when tested with different problem instances, which was the result of the different stopping conditions in the *While* loops of the generated algorithms. Specifically, in KPA1, the pseudocode has loops where it was not possible to anticipate the end. Thus, in general, when generated algorithms contain multiple nested *While* loops, the mode of operation depends on the type of problem instance to be solved, which is an evidence of algorithms that can adapt to different types of problem instances.

With the definition of functions and terminals designed specifically for this automatic generation algorithm, the hyper-heuristic is more compact than that found in the current literature (Burke et al. 2010), because the approach can generate an integrated solution method, which solves the problem as a whole and not "move by move". Some current hyper-heuristics are a concatenation of heuristics where nearly every movement is performed with a different heuristic. While they have good performance in computing, hyper-heuristics such as those just mentioned are specific to a problem, and their generalization to other problems of a different nature is not trivial.

The comparison between generated algorithms with state of the art methods is expressed through the error obtained when solving a set of evaluation instances. This is because the error is measured using the best known solution given by the best algorithm in the state of the art. For the larger instances such error was less than 1 %.

6 Conclusions

In this study, automatically generated algorithms for the BKP are presented. Such algorithms are generated from evolutionary computation, specifically GP. The process follows a hyper-heuristic framework that considers three levels that travel from the innermost layer, the solutions to the BKP, through low-order heuristics that operate on the data structures of the BKP, to the outermost layer, where an evolutionary algorithm creates an algorithm for the BKP.

The results of the study indicated that the algorithms have high computational efficiency and effectiveness in the sense that they rapidly find a solution for each problem instance of the BKP and find a near optimal solution for such problem instances. The tables of results indicate that the algorithms are robust to larger problem instances, i.e., maintaining or improving their performance, but that there is no improvement in quality resulting from the training of the algorithms with more fitness cases. Furthermore, the control of parameters of interest such as the population size and the size of the algorithm reduces the computational effort involved, making the process more efficient as a whole. Regarding the structure of the algorithms, we observed that these do not contain all of the terminals and functions defined in the experiment, which indicates that some components are privileged over others by GP. In addition, within the algorithms, we detected three clearly defined processes, which are pretreating the data, filling the knapsack and refining the solution. In the top three algorithms, the knapsack is always filled with the *Add_Dantzig* terminal within a *While* loop. The pre-treatment and refinement of the knapsack consist of exchanging BKP items according to different criteria that GP favored.

Although the results of this study are promising additional research should be conducted to verify if efficient, effective and robust algorithms also emerge for other combinatorial optimization problems. Since every algorithm starts its operation with an empty initial solution, the resulting algorithms are the combination between constructive and refinement terminals. If a feasible initial solution is always considered then the refinement terminals become more relevant and different algorithmic structures can be explored. In particular, structures corresponding to metaheuristics. Further research in this field would allow to study the automatic generation of metaheuristics.

Acknowledgments The authors would like to thank the Complex Engineering Systems Institute ICM: P-05-004-F, CONICYT: FBO16,

DICYT: 61219-USACH, ECOS/CONICYT: C13E04, STICAMSUD: 13STIC-05.

References

Affenzeller M, Winkler S, Wagner S, Beham A (2009) Genetic algorithms and genetic programming: modern concepts and practical applications, 1st edn. CRC, Chapman and Hall

Ahmed A, Shaikh AW, Ali M et al (2011) Hyper-heuristic approach for solving scheduling problem: a case study. Aust J Basic Appl Sci 5(9):190–199

Alinia A, Vakil B, Badamchi Z et al (2012) Hybrid particle swarm optimization transplanted into a hyper-heuristic structure for solving examination timetabling problem. Swarm Evol Comput 7:21–34. doi:10.1016/j.swevo.2012.06.004

Angelelli E, Mansini R, Grazia S (2010) Kernel search: a general heuristic for the multi-dimensional knapsack problem. Comput Oper Res 37(11):2017–2026

Balas E, Zemel E (1980) An algorithm for large zero-one knapsack problems. Oper Res 28(5):1130–1154

Bienstock D (2008) Approximate formulations for 0-1 knapsack sets. Oper Res Lett 36(3):317–320

Bilgin B, Demeester P, Misir M et al (2012) One hyper-heuristic approach to two timetabling problems in health care. J Heuristics 18(3):401–434. doi:10.1007/s10732-011-9192-0

Burke E, Kendall G, Newall J et al (2003) Hyper-heuristics: an emerging direction in modern search technology. Handbook of metaheuristics, pp 457–474

Burke E, Hyde M, Kendall G et al (2010) A classification of hyper-heuristic approaches. In: Handbook of metaheuristics (international series in operations research & management science, vol 146). Nottingham, UK, pp 449–468

Burke E, Hyde M, Kendall G et al (2012) Automating the packing heuristic design process with genetic programming. Evol Comput 20(1):63–89. doi:10.1162/EVCO_a_00044

Darehmiraki M, Mishmast N (2007) Molecular solution to the 0–1 knapsack problem based on DNA computing. Appl Math Comput 187(2):1033–1037

Demeester P, Bilgin B, De Causmaecker P et al (2012) A hyperheuristic approach to examination timetabling problems: benchmarks and a new problem from practice. J Sched 15(1):83–103. doi:10.1007/s10951-011-0258-5

Fortnow L (2009) The status of the P versus NP problem. Commun ACM 52(9):78–86

Fraser A, Weinbrenner T (1993) GPC++-genetic programming C++ class library. [Online]. ftp://cs.bham.ac.uk/pub/authors/W.B.Langdon/weinbenner/gp.html. Accessed 17 Dec 2013

Garey M, Johnson D (1979) Computers and intractability. A guide to the theory of NP-completeness. A series of books in the mathematical sciences. WH Freeman and Company, San Francisco, Calif

Garrido P, Castro C (2012) A flexible and adaptive hyper-heuristic approach for (dynamic) capacitated vehicle routing problems. Fundam Inform 119(1):29–60. doi:10.3233/FI-2012-726

Gómez JC, Terashima-Marín H (2012) Building general hyper-heuristics for multi-objective cutting stock problems. Computacion y Sistemas 16(3):321–334

Koza J (1992) Genetic programming: on the programming of computers by means of natural selection. The MIT press, Massachusetts

Koza J, Bennett F, Andre D et al (1997) Automated synthesis of analog electrical circuits by means of genetic programming. IEEE Trans Evol Comput 1(2):109–128. doi:10.1109/4235.687879

Koza J, Keane M, Streeter M, Mydlowec W, Yu J, Lanza G (2005) Genetic programming IV: routine human-competitive machine intelligence. Kluwer Academic Publishers, Norwell

Kumar R, Singh P (2010) Assessing solution quality of biobjective 0–1 knapsack problem using evolutionary and heuristic algorithms. Appl Soft Comput 10(3):711–718

Martello S, Toth P (1990). Knapsack problems: algorithms and computer implementations (revised). University of Bologna, Bologna, Italy

Pisinger D (2005) Where are the hard knapsack problems? Comput Oper Res 32(9):2271–2284

Poli R, Langdon W, McPhee N (2008) A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk

Rajni A, Chana I (2013) Bacterial foraging based hyper-heuristic for resource scheduling in grid computing. Future Gener Comput Syst 29(3):751–762. doi:10.1016/j.future.2012.09.005

Sabar N, Ayob M, Qu R et al (2012) A graph coloring constructive hyper-heuristic for examination timetabling problems. Appl Intell 37(1):1–11. doi:10.1007/s10489-011-0309-9

Taghipour H, Rezaei M, Esmaili H (2013) Solving the 0/1 knapsack problem by a biomolecular DNA computer. Advances in bioinformatics

Ye L, Zhang M (2013) Solutions to the 0-1 knapsack problem based on DNA encoding and computing method. J Comput 8(3):669–675