Solving Knapsack Problems with Attribute Grammars

Michael O'Neill, Robert Cleary, and Nikola Nikolov

Biocomputing & Developmental Systems Group
University of Limerick, Ireland
Michael.ONeill@ul.ie, Robert.Cleary@ul.ie, Nikola.Nikolov@ul.ie

Abstract. We present a work in progress describing attribute grammar approaches to Grammatical Evolution, which allow us to encode context-sensitive and semantic information. Performance of the different grammars adopted are directly compared with a more traditional GA representation on five instances of an NP-hard knapsack problem. The results presented are encouraging, demonstrating that Grammatical Evolution in conjunction with alternative grammar representations can provide an improvement over the standard context-free grammar, and allow Grammatical Evolution to drive a constraint based search.

1 Introduction

Combinatorial optimisation problems, such as knapsacks problems, represent an important class of problems in computer science that have a number of real world applications such as resource allocation and cargo loading. Given the difficulty of these, in some case NP-hard problems, stochastic heuristic methods such as evolutionary algorithms have been investigated as approaches to find approximations to optimum solutions. A characteristic of these problems is the requirement of their solutions to meet a number of constraints, which for efficiency, should be respected by the search algorithm in order to prevent the existence of infeasible solutions that violate one or more constraints. In this study we demonstrate the benefits of adopting an attribute grammar with Grammatical Evolution in tackling five knapsack problem instances. The advantage of the attribute grammar over a standard context-free grammar is in its ability to directly encode the constraints of the problem instance as part of the production rule specification, and to encode semantic information on the state of a developing solution during the genotype-phenotype mapping process, which can be used to test for constraint violation during development.

The remainder of the paper is structured as follows. An introduction to the class of combinatorial optimisation problems that encompass knapsacks are introduced in Section 2, followed by a short description of Grammatical Evolution in the context of knapsack problems in Section 3. Attribute grammars and their application to knapsack problems are discussed in Section 4 followed by details on the experimental setup in Section 5. Finally the results are presented in Section 6 and conclusion and future work outlined in Section 7.

2 Knapsack Problems

This section provides a brief overview of the family of knapsack problems, explaining the principles common to each class of problem. We then go on to provide the motivation for this choice of problem, followed by an explanation of the particular class of knapsack problem that is being tackled in this paper.

2.1 Brief Overview of the Problem

Knapsack problems refer to a class of combinatorial optimisation problems with analogy to filling a knapsack with items of varying worth. The principle behind these problems, is to add items to a container, while respecting a weight constraint, such that we maximise the objective function (a measure of the combined profit of the items).
As this paper concerns a category of knapsack problems entitled *Zero-One (0/1)*, we will concentrate on a description of these in particular. For a good review of knapsack problems in general, and their categorisation; we refer the reader to Pisinger's Ph.D thesis [1, 2], which describes the family of Knapsack problems, and provides a mathematical definition of each class. Also recommended is Martello and Toth's book which is widely regarded as a canonical text for knapsack problems [3].
Different types of Knapsack problem occur, depending on the number of knapsacks; the number of items, and the way in which items can be added to a knapsack. Given that there is a set of possible items, and our goal is to choose a subset of these, the term 0/1 refers to a problem whereby an item can only be chosen for inclusion in the knapsack once (i.e., an item is either in or not in). You cannot fill a knapsack with $n > 1$ of the same item to maximise the profit. At this point we wish to introduce the term *0/1 Compliant* to refer to a solution which satisfies this 0/1 property.
Breaking the 0/1 property allows for *Bounded* Knapsack problems. That is, an arbitrary percentage of each item can be chosen in order to attain the largest objective value. *Multiple-Choice* Knapsack problems occur when items are chosen from disjoint sets, and the most general form of problem, entitled *Multi-Constrained* or *Multi-Dimensional* knapsack problem, occur when we have multiple knapsacks with independent constraints and weights on each; into which the same set of items must be placed.

2.2 Motivation behind Choice of Problem

We have chosen the 0/1 Multi-Dimensional Knapsack Problem (0/1 MKP) to demonstrate the work of this paper. Motivations behind the choice of this particular class of knapsack problem include the fact that the 0/1 MKP is a well known NP-hard problem. Also, there exists, a repository of instances of 0/1 MKP's available form the OR Library [4][1] which acts as a suitable experimental

[1] http://mscmga.ms.ic.ac.uk/info.html

benchmark. Previous researchers in the field of Evolutionary Computation have applied their work to this problem which provides a point of reference to the value of this work [5–8]. Of more relevance to this paper, the problem lends itself to demonstrate how the use of attribute grammars as opposed to context-free grammars can give context to the current derivation step (See Sect.4).
Finally, we approach the problem as an abstract academic problem, with respect to *proof of concept*, but acknowledge the possibilities it provides for application to practical problems, such as; cutting stock, cargo loading, resource allocation in distributed computing, and integer programming and budget control.

2.3 The 0/1 MultiDimensional Knapsack Problem

The 0/1 MultiDimensional Knapsack Problem (or Multi-Constrained Knapsack problem) deals with a problem, whereby we have a number of knapsacks to be filled with a set of items. Each knapsack has a maximum capacity or *weight-constraint*. We must select a subset of the set of all items (the *vector of items*), for inclusion in all knapsacks; such that the combined weight of this chosen subset, doesn't violate the weight-constraint of any of the knapsacks. As an added condition of the problem; the weight of an item is variable, and it is determined with respect to which knapsack it is included in. This will be termed the *relative-weight* of an item. As a consequence, a possible solution or chosen vector of items will have varying weight in each knapsack.
Although the weight of an item is variable, an item has a fixed value or profit. Thus the goal or objective for this problem, is to select a vector of items, with maximal profit or worth, whilst respecting the weight-constraints of all knapsacks. The problem can be formulated as

$$maximise \sum_{j=1}^{n} p_j x_j \quad (1)$$

$$subject\ to \sum_{j=1}^{n} w_{ij} x_j \leq c_i, \qquad i = 1 \ldots m \quad (2)$$

$$x_j \in \{0, 1\}, \qquad j = 1 \ldots n \quad (3)$$

Where, p_j refers to the profit, or worth of item j, x_j refers to the item j, w_{ij} refers to the relative-weight of item j, with respect to knapsack i, and c_i refers to the capacity, or weight-constraint of knapsack i. There are present $j = 1 \ldots n$ items, and $i = 1 \ldots m$ knapsacks.
The objective function (equation 1) tells us to find a subset of the possible items (ie. the vector of items); where the sum of the profits of these items is maximised, according to constraints presented in equation 2. Equation 2 states, that the sum of the relative-weights of the vector of items chosen, is not to be greater than the capacity of any of the m knapsacks. Equation 3 refers to the notion that we wish to generate a vector of items, of size n($j = 1..n$ items), whereby a 0 at the

i^{th} index indicates that this item is not in the chosen subset and a 1 indicates that it is.
As described in [6], it is also worth thinking of the problem as a matter of resource allocation. That is, we have m resources (knapsacks) and n tasks. Each resource has a budget or knapsack capacity C_i, and W_{ij} represents the consumption of resource i by task j. Thus, $task_n$ may then have a different resource-consumption, depending on which of the m resources it is applied to (ie. it's *relative weight*). The objective then, is to select a set of tasks to be applied to all resources simultaneously, such that, the budgets of each resource are respected, and the consumption of resources is optimised.
The following section proceeds to describe the limitations faced by Grammatical Evolution, when generating solutions to adhere to the constraints of problems such as the 0/1 Multi-Dimensional knapsack problem.

3 Grammatical Evolution and Context Free Grammars

Grammatical Evolution (GE)[9–11] can be thought of as a system that evolves sentences in the language described by a context-free grammar (CFG). A *sentence* is a string of terminal symbols resulting from a derivation-sequence, or the application of n production rules to the start symbol S. Thus, it can be thought that what GE really evolves; is the sequence of productions we apply to the start symbol S in order to arrive at the sentence λ. That is, GE can be thought of as a process of evolving derivation-sequences over a grammar.
However, a limitation to such derivation-sequences is the tool used to generate the sentences (ie. *the grammar*). As standard GE adopts a CFG, there is no way to provide context-sensitive or semantic information when carrying out a derivation-step. As the name suggests CFG's cannot express a language in which; legal phrases, depend on the context in which they are applied.
That is, *a phrase* can be thought of as a portion of a derivation-tree descended from a single non-terminal symbol. We will refer to a phrase which contains a terminal symbol as a *a terminal-producing production*. Consider the following example of a CFG to describe a language consisting of a set of items $\{i_1, \ldots, i_n\}$. Adopting the notation of Knuth [12] we define our context free grammar for an n item knapsack as follows:

$$
\begin{aligned}
&S \rightarrow K \\
&K \rightarrow I \\
&K \rightarrow IK \\
&I \rightarrow i_1 \\
&\vdots \\
&I \rightarrow i_n
\end{aligned}
$$

Beginning from the start symbol S a sentence in the language of knapsacks is created by application of productions to S such that only terminal symbols remain; yielding a string from the set of items $\{i_1, \ldots, i_n\}$.

Consider the problem of generating such a string for a 0/1 knapsack problem as defined in the previous section. GE essentially carries out a left-most derivation, according to the grammar specified. The following derivation-sequence illustrates the point at which a CFG fails to be able to uphold context-specific information.

$$S \rightarrow K \rightarrow IK \rightarrow i_3K \rightarrow i_3IK \rightarrow i_3??$$

What this derivation-sequence provides is a *context*. That is, given the context that i_3 has been derived, the next derivation-step must ensure that i_3 is not produced again. Additionally, if we enforce a constraint that limits the total weight of the knapsack, a derivation-sequence of a CFG does not allow us to determine if this constraint has been violated until the derivation is complete. A CFG has no method of encoding this context-sensitive information.
The following section describes how we can overcome the limitations of CFG's and give context to the current derivation step, by employing an attribute grammar to encode this information.

4 Attribute Grammars for Knapsacks

Attribute grammars were first introduced by Knuth [12], as a method to extend CFGs by assigning attributes (or pieces of information), to the symbols in a grammar. Attributes can be assigned to any symbol of the CFG, whether terminal or non-terminal, and are defined (given meaning) by functions associated with productions in the grammar. These shall be termed the *semantic functions*. Attributes can take the form of simple data (integers), or more complex data-structures such as lists, which append to each symbol of the grammar. Attributes are evaluated in two ways. In the first, the value of an attribute is determined by the value of the attributes of child nodes. That is, it is *synthesised* or made up of it's children's values. In the second, the value of an attribute is determined by information passed down from parent nodes. That is, it is evaluated based on what is *inherited* down from parent nodes. Information originates either from the root or leaf nodes of the tree, which generally provide constant values from which, the value of all other nodes in the tree are synthesised or inherited.

4.1 An Attribute Grammar for 0/1 Compliance

Consider the following attribute grammar specification to show how attributes can be used to preserve 0/1 compliance when deriving strings in the language of knapsacks (*see Sect* 2). This attribute grammar is identical to the earlier CFG, with regard to the syntax of the knapsacks it generates. The difference here being the inclusion of attributes associated with both terminal and non-terminal symbols, and their related *semantic functions*. As each symbol in the grammar maintains it's own set of attributes, we use a subscript notation to differentiate between occurrences of like non terminals.
Following the notation of Knuth [12], we have appended the following attributes to the previous CFG grammar:

items(K): A synthesised attribute that records all the items currently in the knapsack (ie. *items which have been derived thus far*).
item(I): A string representation, identifying which physical item the current non-terminal will derive. For example $item_1$ is represented as the string "i_1".
notInKnapsack?(i_n): A boolean flag, indicating whether the 0/1 property can be maintained by adding this item (ie. given the current derivation, has this item been previously derived?). This is represented as a string-comparison of *item(I)* over *items(K)*.
The following gives a description of such an attribute grammar, and provides an example to illustrate how it can be used to drive a context-specific derivation

$$
\begin{array}{ll}
S \rightarrow K & \\
K \rightarrow I & items(K) = items(K) + item(I) \\
 & \\
K_1 \rightarrow IK_2 & items(K_1) = items(K_1) + item(I) \\
 & items(K_2) = items(K_1) \\
 & \\
I \rightarrow i_1 & item(I) = \text{“}i_1\text{”} \\
 & \textbf{Condition}: \quad if(notinknapsack?(i_1)) \\
\vdots & \\
I \rightarrow i_n & item(I) = \text{“}i_n\text{”} \\
 & \textbf{Condition}: \quad if(notinknapsack?(i_n))
\end{array}
$$

Consider the above attribute grammar, when applied to the following derivation-sequence:

$$S \rightarrow K \rightarrow IK \rightarrow i_1K \rightarrow i_1IK \rightarrow i_1(i_\lambda \in \{i_2 ... i_n\})K \rightarrow ...$$

At the point of mapping I given the above context, it can be seen from the above semantic functions that it's *items(I)* attribute will be evaluated to "i_λ" if the *notinknapsack?()* condition holds. Following this the root node will have it's *items(K_1)* updated to include "i_λ" which can from then on be passed down the tree by the inherited attribute of *items(K_2)*. This in turn allows for the next *notinknapsack()* condition to prevent duplicate items being derived. The next section follows to provide a deeper example, which shows how we can include the evaluation of weight-constraints at the point in a derivation where we carry out a terminal-producing production.

4.2 An Attribute Grammar for Constraints Checking

The previous attribute grammar provided a description of attributes for determining if a terminal-producing production would derive a duplicate item. We now present an attribute grammar which builds upon that; adding the following attributes:
lim(S): A global attribute containing each of the m knapsacks' weight-constraints. This can be inherited or passed down to all nodes.
lim(K): As lim(S) just used to inherit to each K_2 child node.
usage(K): A usage attribute, records the total weight of the the knapsack to

date. That is, the weight of all items which have been derived at this point.
weight(K): A weight attribute, used as a variable to hold the weight of the item derived by the descendant I to this K.
weight(I): A synthesised attribute, made-up of the descendant item's physical weight.
weight(i_n): The physical weight of item i_n(*the weight of item* i_n *as defined by the problem instance*).

The corresponding attribute grammar is given below with an example showing how it's attributes are evaluated.

$$
\begin{array}{ll}
S \rightarrow K & lim(K) = lim(S) \\
 & \\
K \rightarrow I & weight(K) = weight(K) + weight(I) \\
 & \textbf{Condition}: \quad if(usage(K) + weight(I) <= lim(K)) \\
 & \quad items(K) = items(K) + item(I) \\
 & \\
K_1 \rightarrow IK_2 & weight(K_1) = weight(K_1) + weight(I) \\
 & \quad items(K_1) = items(K_1) + item(I) \\
 & \quad usage(K_1) = usage(K_1) + weight(I) \\
 & \textbf{Condition}: \quad if(usage(K_1) < lim(K_1)) \\
 & \quad lim(K_2) = lim(K_1) \\
 & \quad usage(K_2) = usage(K_1) \\
 & \quad items(K_2) = items(K_1) \\
 & \\
I \rightarrow i_1 & item(I) = \text{“}i_1\text{”} \\
 & \textbf{Condition}: \quad if(notinknapsack?(i_1)) \\
 & \quad weight(I) = weight(i_1) \\
\vdots & \\
I \rightarrow i_n & item(I) = \text{“}i_n\text{”} \\
 & \textbf{Condition}: \quad if(notinknapsack?(i_n)) \\
 & \quad weight(I) = weight(i_n)
\end{array}
$$

In terms of the problem being solved, *lim(K)* is actually a list of constraint-bounds for each of the m knapsacks. Similarly, *items(K)*, is a list of the items which have currently been derived by the GE mapping process. For clarity of explanation, the following example will assume a single knapsack weight-constraint, but the more complicated problem can be extracted by altering the below conditions to have *lim(K)* as an array of constraint-bounds as opposed to a single integer value.
Figure 1 shows the synthesised and inherited message passing involved in evaluating derivation trees for the above attribute grammar. The figure shows a fully decorated tree, but also includes a illustrative explanation of the use of conditions for the grammar. We can see, that initially the global limit is passed down to K by the first semantic function. From the grammar, we can see that following this, the first three semantic functions of K are evaluated before a

condition checks to see that we haven't violated a weight-constraint[2]. Passing this allows for inheriting values down the tree by the second three semantics functions (otherwise we would have remapped K via another production and repeated the process).

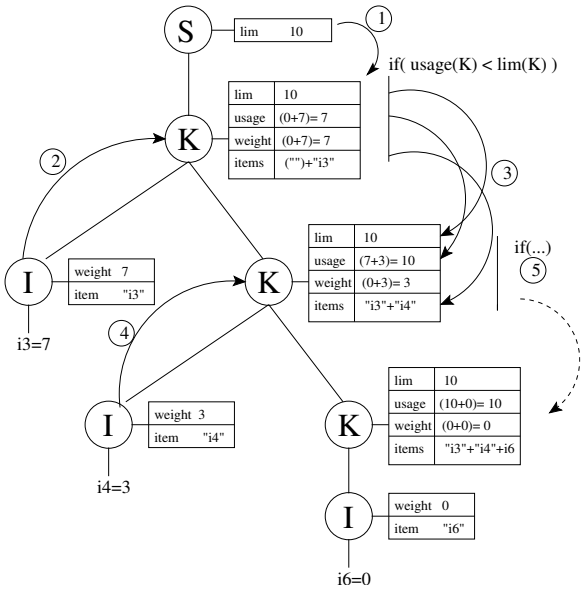

Fig. 1. Diagram showing synthesised and inherited message-passing for evaluating attributes in the derivation tree of an attribute grammar.

Having understood the method by which the attribute grammar gives context to the current derivation-step, with respect to knapsack capacities; the next section follows to give an explanation of the experimental approach taken.

5 Experiments

We adopt standard experimental parameters for GE, however as an *individuals processed* based comparison is taking place, the number of generations varies with each problem. The benchmark GA system uses a population size of $\mu =$ 50 running for up to 2000 generations. We match the number of individuals processed by altering our generations accordingly. We use a population size of μ =500. For the results presented, dividing the individuals processed by 500 gives the generation at which we report results. We adopt a variable length one-point crossover probability of 0.9, bit mutation probability of 0.01, and roulette

[2] For clarity, we assume that the *notInKnapsack(i_3)* condition has passed and the its values have synthesised up the tree.

selection. A steady-state evolutionary process is employed, whereby a generation constitutes the evolution and attempted replacement of $\mu/2$ children into the current population. Replacement occurs if the child is better than the worst individual in the population.
The initial population of variable-length individuals were initialised randomly, with an average length of 20 codons, and standard-deviation of 5 codons from average. Standard 8-bit codons are employed, and GE's wrapping operator is turned off. For each experiment 30 runs were performed.

5.1 Experimental Setup

For experimental purposes, standard GE is tested against two variants of GE which use differing grammar representations. These systems will now be discussed in terms of their grammar characteristics and the solutions they generate. That is, due to the nature of the problem, a solution may be either *feasible* or *infeasible*. The latter refers to a solution which breaks one or more constraints of the problem (ie. is either non 0/1 compliant or violates a weight-constraint. In all systems allowing the generation of infeasible solutions, yields a fitness penalty. This fitness penalty is discussed in the next section.

System 1: GE This system is a standard GE setup, using the above parameters and algorithmic configuration. No attributes or semantic functions exist to give context to the current derivation-step. Infeasible solutions and non-0/1 compliant solutions can be generated. Possible solutions are examined for 0/1 compliance prior to their evaluation, failure of this test results in penalty to the worst possible fitness. The system uses a standard CFG as defined in Section 3.
System 2: GE+0/1 This approach is an extension of system 1. The standard GE mapping process is adapted using the attribute grammar of Section 4.1, so as to maintain 0/1 compliant solutions. With regards this attribute grammar, the attribute *items(K)* is employed to do this by passing around the previously mapped items to each derivation step. A condition on the *notInKnapsack?()* attribute, allows the identification of a previously mapped item in which case we read the next codon and re-map the offending Non-Terminal. This approach allows infeasible solutions, but maintains the 0/1 property.
System 3:- GE+AG This approach is an extension of system 2. This system uses the full attribute grammar as defined in the Section 4.2. As well as guaranteeing the 0/1 property, it carries out a constraints check on all m knapsacks, for a terminal-producing production. At the point of mapping, 0/1 compliance is ensured, and another test is carried out to ensure that adding the terminals in this production doesn't violate any of the m weight-constraints. If a weight-constraint is violated, we read the next codon and re-map the offending Non-Terminal. This system only allows feasible solutions.

We perform a direct comparison based on the results obtained by Khuri et al. for five knapsack problem instances [6]. This benchmark provides a comparison with a more traditional Genetic Algorithm representation. As we compare our work

against a different evolutionary algorithm, we maintain our system in terms of standard GE parameters, and perform the comparison in terms of the number of individuals processed. It is also worth noting the difference in representations. The benchmark system uses a fixed-length bit vector representation consistent with equation 3 of Section 2.3, whereas our representation is that of a variable-length genome mapping to a derivation-sequence of a length which is equal to or less than the genome length depending on constraint satisfaction.
As the benchmark system presents a GA approach which uses a graded penalty term to penalise infeasible solutions, according to the amount by which they break constraints; we also provide a comparison to see the effect of this with respect to our systems. That is, two independent runs are performed for each problem whereby penalisation of infeasible solutions in the graded-penalty systems is relative to how badly weight-constraints are violated; and in the other, penalty to the worst possible fitness is performed. Note that, due to the steady-state system this results in the removal of such infeasible solutions from the search space over time.
Two experimental measures of *mean-best fitness* and *percentage of runs yielding and optimum solution* are employed.

6 Results

Results for five problem instances[3] are presented in Tables 1 and 2. A comparison of the performance of the attribute grammar approach on all problem instances clearly demonstrates its superior performance to the context free grammar.
Table 1 shows both attribute grammar systems outperforming the standard GE for all problems, with GE+AG achieving the best results in all cases. The graded-section of this table show improvement for easier problems but this declines to disimprovement for harder problems.
Table 2 again shows the attribute grammar systems outperforming the standard GE in terms of the number of optimum solutions found for the earlier problems, but the effects for later problems are indecipherable as no optimum is found in the given number of individuals processed. The graded-section shows a disimprovement in the easier problem but an improvement is seen for GE+0/1 over the knap20 problem.

7 Conclusions & Future Work

Although the results presented do not appear competitive with the benchmark system which uses a more traditional GA representation, we can draw the conclusion that attribute grammars show promising results with respect to enhancing GE's capabilities to tackle problems with constraints. These are early investigations, however, and future work will look at a number of different knapsack

[3] These problem instances are available from
http://mscmga.ms.ic.ac.uk/jeb/pub/mknap1.txt

Problem	**knap15** *15-10-4015*	**knap20** *20-10-6120*	**knap28** *28-10-12400*	**knap39** *39-5-10618*	**knap50** *50-5-16537*
IndProcessed	5000	10,000	50,000	100,000	100,000
MeanBst-Khuri	4012.7	6102	12374.7	10536.9	16378.0
MeanBst-GE					
GE	3672.66	5778.0	11482.83	9320.96	14520.36
GE+01	3946.33	5967.66	12030.16	9662.86	14949.9
GE+AG	3982.66	6061.66	12129.5	9861.0	15228.7
MeanBst-GE Graded					
GE	3675.0	5787.83	11661.33	9268.73	14368.36
GE+01	3947.33	5976.66	12030.16	9597.4	14949.76

Table 1. Comparing Mean-Best Fitness achieved after n Individuals processed. Showing the effects of Attributed Grammars over GE, and the effect of further adding a Graded Penalty function.

Problem	**knap15** (15-10-4015)	**knap20** (20-10-6120)	**knap28** (28-10-12400)	**knap39** (39-5-10618)	**knap50** (50-5-16537)
IndProcessed	5000	10,000	50,000	100,000	100,000
Runs Opt-Khuri	83%	33%	33%	4%	1%
Runs Opt-GE					
GE	3%	0%	0%	0%	0%
GE+01	16%	0%	0%	0%	0%
GE+AG	26%	33%	0%	0%	0%
Runs Opt-GE Graded					
GE	0%	0%	0%	0%	0%
GE+01	13%	3%	0%	0%	0%

Table 2. Showing the effect Attributed Grammars have over percentage of runs achieving an optimum solution, and also the effect of further adding a graded penalty function gives.

instances and a deeper analysis of this representation. For example, due to the nature of the different population sizes, some of our results presented are too early in the evolution process to be deemed a fair comparison. We also note the difference in representation as an area for research. If we alter the attribute grammar to impose a fixed-length phenotypical structure to reflect a bit-vector similar to the GA representation adopted in the benchmark study, would this have a positive impact on performance? In addition, we will investigate the possible extension of this approach to different classes of combinatorial optimisation problems.

References

1. Pisinger, D. (1995) *Algorithms for Knapsack Problems.* Ph.D. thesis, DIKU, University of Copenhagen, Report 95/1.
2. Pisinger, D., Toth, P. (1998). Knapsack Problems, in D.Z. Du, P. Pardalos (eds.) *Handbook of Combinatorial Optimization*, Kluwer, pp. 1 -89.
3. Martello, S., Toth, P. (1990). *Knapsack Problems.* J. Wiley & Sons, Chicester, 1990.
4. Beasley, J.E. (1990). OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society* Vol. 41 No. 11, pp. 1069-1072.
5. Chu, P.C. and Beasley, J.E (1998).A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics* 4:63-86.
6. Khuri, S., Back, T., and Heitkotter, J. (1994).The zero/one multiple knapsack problem and genetic algorithms. In Deaton, E. et al., editors, *Proceedings of the 1994 ACM symposium of Applied Computation,* pages 188-193, ACM Press, New York.
7. Bruhn, P., Geyer-Schulz, A. (2002). Genetic Programming over Context-Free Languages with Linear Constraints for the Knapsack Problem: First Results. *Evolutionary Computation*, Vol. 10, No. 1, Spring 2002.
8. Ratle, A. and Sebag, M. (2000). Genetic Programming and Domain Knowledge: Beyond the limitations of grammar-guided Machine Discovery. *In M. Schoenauer et al., ed, Parallel Problem Soving from Nature, PPSN-VI,* p. 211-220. Springer Verlag, LNCS 1917.
9. O'Neill, M., Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers.
10. O'Neill, M. (2001). *Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution.* PhD thesis, University of Limerick, 2001.
11. O'Neill, M., Ryan, C. (2001). Grammatical Evolution, *IEEE Trans. Evolutionary Computation.* 2001.
12. Knuth, D.E. (1968). Semantics of Context-Free Languages. *Mathematical Systems Theory*, Vol. 2, No. 2. Springer-Verlag.