# An Attribute Grammar Decoder for the 01 MultiConstrained Knapsack Problem

Robert Cleary and Michael O'Neill

University of Limerick, Ireland
Robert.Cleary@ul.ie, Michael.ONeill@ul.ie

**Abstract.** We describe how the standard genotype-phenotype mapping process of Grammatical Evolution (GE) can be enhanced with an attribute grammar to allow GE to operate as a decoder-based Evolutionary Algorithm (EA). Use of an attribute grammar allows GE to maintain context-sensitive and semantic information pertinent to the capacity constraints of the 01 Multiconstrained Knapsack Problem (MKP). An attribute grammar specification is used to perform decoding similar to a first-fit heuristic. The results presented are encouraging, demonstrating that GE in conjunction with attribute grammars can provide an improvement over the standard context-free mapping process for problems in this domain.

## 1 Introduction

The NP-Hard 01 Multiconstrained Knapsack Problem (MKP) can be formulated as;

$$maximise \qquad \sum_{j=1}^{n} p_j x_j \tag{1}$$

$$subject\ to \qquad \sum_{j=1}^{n} w_{ij} x_j \qquad \leq c, \tag{2}$$

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

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 exist $j = 1 \ldots n$ 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.

Exact methods such as Branch and Bound have been found as good approximation algorithms to the single-constrained knapsack problem [1], but however Evolutionary Algorithms (EAs) have been found to be most appropriate in solving large instances of the MKP for which exact methods are too slow. As a result EAs, and in particular, decoder-based EAs have been heavily studied in application to the MKP [2–8]. Their advantage over the more traditional direct representation of the problem, is their ability to always generate and therefore carry out evolution over feasible candidate solutions, and thus focus the search on a smaller more constrained search space [9, 10]. The best EAs for the MKP that we are aware of utilise problem-specific domain knowledge to carry out repair and optimisation to maintain feasible solutions [11, 12, 5].

These EAs have been developed specifically for solving the MKP, and are based heavily on domain knowledge of the problem and efficiency to solution time via locally-optimised initialisation and search techniques. It is not the focus of this paper to attempt to compete with such algorithms, rather, in this instance; we wish to examine the ability of Grammatical Evolution's (GE's) mapping process to be transformed to the role of a decoder for constrained optimisation problems. More specifically, we use constrained optimisation problems as a test-bed to demonstrate how attribute grammars allow the extension of GE to context-sensitive problem domains. As a side effect, we also see the possibility to further our analysis of the internal workings of GE, through merging research in the methods of analysis found within the field of decoder-based EAs. Core to the functioning of such decoder-based EAs is a genotype-phenotype mapping process, and methods have been developed for the effective analysis of the workings of such mapping processes.

The remainder of the paper is structured as follows. An introduction to decoder-based EAs from the literature is presented 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 Decoder Approaches from the literature

The previous section outlined the knapsack problem as that of a constrained optimisation problem. From our literature review we divide the various approaches into two categories; *infeasible*, and *feasible-only*. From this survey we encountered many successful works from both approaches, with Raidl's improved GA [5] being the best infeasible approach, outperforming Chu and Beasley's [11] GA by what is reported to be a non-deterministic local optimisation strategy. Of the feasible-only approaches, the problem space decoder based EA of Raidl [12], marginally outperforms Gottlieb's study of *permutation-based EAs* in [2].

### 2.1 Infeasible Solutions

The allowance of infeasible solutions within the evolving population of the EA stems from what [9] and [12] refer to as a *direct* representation. That is, chromosomes encode for a set of items to be included in the knapsack. Each gene represents a corresponding item. The most typical use of this approach is the binary bit-string representation of size $n$ where a 1 at the $i^{th}$ index indicates that this item is to be included in the knapsack.
As pointed out in [8], it is important with this kind of approach to ensure that the infeasible solution doesn't end up in the final population, or result in being awarded a fitness better than a feasible solution. Khuri and Olsen [13, 14], both report only moderate success rates with such approaches; and as such we focus our attention on the use of decoders in providing feasible-only solutions.

### 2.2 Feasible-Only Solutions

The simple structure of the MKP is often approached by applying the search algorithm to the space of possible solutions, mapped out by $P \in \{0,1\}^n$. A common alternative to this is the use of an EA which works in some other search space which maps into $F$ the feasible subset of $P$ [9]. The entity which does such a mapping is generally referred to as a decoder, and guarantees the generation of solutions that lie within $F$, and often further constrain the search to more promising regions of $F$.
In principle, all decoders work the same way. A decoder can be thought of as a builder of legal knapsacks. It works by attempting to add items to a knapsack, whereby each attempted addition of an item is governed by a capacity-constraints check. That is, items are added as long as no capacity constraint is violated by it's addition.
Decoders differ; in their use of problem specific heuristic information and how they interpret the genome. That is, EAs which choose the decoder approach to constrained optimisation problems, utilise a representation which feeds the decoder's internal knapsack generation algorithm. This representation is generally one of two different classes: *a*) The chromosome representation is a mapping to some permutation of the set of items that implicitly defines the order by which the decoder attempts to build a knapsack. Such EAs are referred to as *order-based*, and have been exhaustively studied in [2–6] and [7, 8] where the general conclusion is that this is a very effective approach; *b*) The chromosome uses a symbolic representation, where genes represent actual items; and the decoder works by dropping items which violate capacity constraints. In this case, evolutionary operators must incorporate the decoder heuristic so as to allow only the generation of feasible solutions. Hinterding [7], introduces such an EA as an order-independent mapper and reports it to perform well, although it is outperformed by an order-based approach.
With either approach, the introduction of a many-to-one genotype-phenotype mapping may occur. Raidl et al [3] refers to this as *heuristic bias*, whereby the stronger the restriction of the search space to promising regions of the feasible

search space $F \subset P \in \{0,1\}^n$, the stronger the heuristic bias. The building of a legal knapsack is governed by termination at the point of a capacity violation. Thus, many of the same genotypes may decode to the same phenotype as all of their genetic material may not be allowed to be used with such decoder termination.

Although working with the simpler single-constrained knapsack problem, Hinterding reports that a redundant mapping from genotype to phenotype gave better results [8]. This is similarly supported in [3, 4] where it is also observed that although desirable too much redundancy in the decoder may result in degradation of performance. The following section will describe the Grammatical Evolution EA, and demonstrate how it lends itself to act as a decoder for constrained optimisation problems via the introduction of an attribute grammar into the genotype-phenotype mapping process.

## 3 Grammatical Evolution

Grammatical Evolution (GE) [15], is an evolutionary algorithm that can evolve computer programs in any language, and can be considered as a form of grammar-based genetic programming. Rather than representing the programs as parse trees, as in standard GP [16, 17], a variable-length linear genome representation is used. Fig. 1 provides an illustration of the mapping process over a simple example CFG.

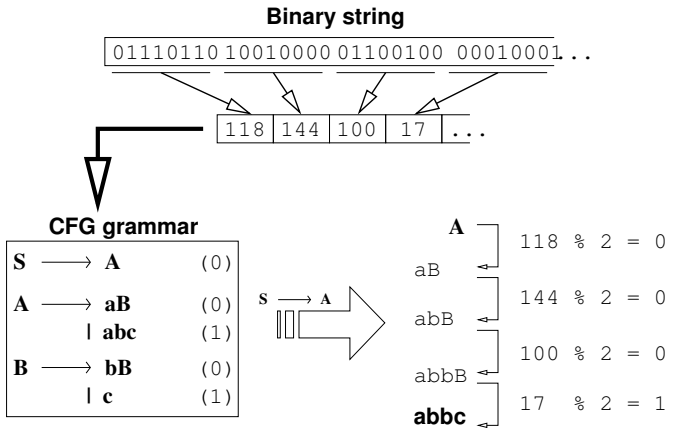


**Fig. 1.** An illustration of GE's genotype-phenotype mapping process' operation.

As illustrated, GE uses the CFG as a *phrase-structure generative grammar*, whereby, rules of the grammar - outline the structure by which syntactically cor-

rect sentences of the language can be derived[1]. It can be seen that the grammar specification of Fig. 1 defines *a language*. This language, written $L(G)$, determines the set of legal (*or syntactically correct*) sentences, which can be generated by application of the grammar's rules. For example, the grammar within the illustration defines the language $L(G) = ab^+c$ : the set of strings starting with the terminal-symbol 'a' - ending in the terminal-symbol $c$; and having one-or-more of the terminal-symbol 'b' in between (Note: the $^+$ symbol denotes, *one-or-more*). The non-terminal symbols 'A' and 'B' define the phrase-structure of the language. They define *A-phrases* and *B-phrases*, from which the language is contained. These would be similar to constructs such as *noun-phrases* in spoken language, or for example, a *boolean-expression phrase* from the abstract syntax of a programming language. In terms of the example grammar, a syntactically valid *A-phrase* contains an 'a' followed by a *B-phrase*. A recursive definition of the *B-phrase* thereafter, defines the previously stated language of the grammar. In this way, the structure of the syntax of an entire language can be defined in a concise and effective notation.

Rules of the grammar are referred to as *production-rules* and as such $A \rightarrow aB$, can be read as, "$A$" *produces* "$aB$". Similarly it can be said that "$aB$" is derived from $A$. Such a production is said to yield a derivation in the *sentential-form*, where by a completed derivation results in a sentential form consisting solely of terminal symbols - a sentence of the language.

The GE mapping process works, by first constructing a map of the grammar, such that left-hand-side non-terminals are used as a key to a corresponding right-hand-side list of rules(*the index of which are specified in parenthesis*). Production-rules are chosen, then, by deriving the production at the index of the current non-terminal's rule-list as specified by the following formula:

$$Rule \;=\; CodonValue \,\%\, Num.\, Rules$$

where % represents the modulus operator. (So as not to detract from the focus of the paper, we defer the interested reader to the canonical texts of GE for a explanation of the intricacies of the mapping process' algorithm [15, 19–21].)

### 3.1 CFG Decoder Limitations

In considering GE as a decoder for knapsack problems then, we wish to use the mapping process to *decode* a genotype, into sentences of the language of knapsacks. That is, we require a CFG definition to represent the language of feasible knapsacks. Let us now consider the viability of such a grammar-based decoder as is afforded by the standard GE mapping process. We can define a grammar for an $n$ item knapsack problem as follows:

$$S \rightarrow K$$
$$K \rightarrow I$$
$$K \rightarrow IK$$

[1] Although GE uses a grammar in Backus-Naur-Form (BNF) - for clarity of explanation of the subsequent attribute grammars, we subscribe to the notation of Knuth [18] to do the same; whereby a $\rightarrow$ denotes a production, as opposed to ::= in BNF.

$$I \rightarrow i_1$$
$$\vdots$$
$$I \rightarrow i_n$$

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 however, the problem of generating such a string for a 01 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. Re-deriving an $i_3$ violates the semantics of the language of 01 knapsacks. A CFG has no method of encoding this context-sensitive information and hence, cannot be used as a decoder to decode chromosomes to feasible knapsack solutions The answer to these limitations lies in the power of attribute grammars, which allow us to give context to the current derivation step. By employing an attribute grammar as the generative power of the mapping process we can extend GE to become a decoder for feasible-only candidate solutions.

## 4 Attribute Grammars for Knapsacks

Attribute grammars (AGs) were first introduced by Knuth [18], 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 (eg. integers), or more complex data-structures such as lists, which append to each symbol of the grammar. In terms of AGs it's best to think of a derivation by it's tree representation where the root is $S$ and it's children the symbols of the applied production. A portion of a derivation-tree descended from a single non-terminal node comprises the context of a phrase. A sentential-form is the set of nodes *directly* descended from such a non-terminal. Also, the term *terminal-producing production* will be used to refer to a sentential-form which contains one or more terminals.

Attributes are thus pieces of data appended to nodes of the tree, and can be evaluated in one of two ways. In the first, the value of an attribute is determined by the value of the attributes of child nodes. That is, the evaluation of a parent attribute can be *synthesized* or made up of it's child's attribute values. In the second, the value of an attribute is determined by information passed down from parent nodes. That is, a child's attribute is evaluated based on information which is *inherited* down from parent nodes. In either case, attributes of a node *can* be evaluated in terms of other attributes of that same node. Information however, originates either from the root node $S$ or leaf nodes of the tree, which generally

provide constant values from which, the value of all other nodes in the tree are synthesized or inherited.

### 4.1 An Attribute Grammar for 01 Compliance

Consider the following attribute grammar specification to show how attributes can be used to preserve 01 compliance when deriving strings in the language of knapsacks. 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 [18], we have appended the following attributes to the previous CFG grammar:
***items(K):*** A synthesized 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 $item(I) =$ "$i_1''$ where that $I$ derives or produces $i_1$ of the problem.
***notInKnapsack?($i_n$):*** A boolean flag, indicating whether the 01 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 "$i_\lambda$" if the *notinknapsack?()* condition holds. Following this the root node will have

it's *items($K_1$)* updated to include "$i_\lambda$" 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

Further attributes can be added, in order to extend the context-sensitive information captured, during a derivation. The following outlines these attributes and their related semantic-functions in a full AG specification, which maintains both 01 and constraint-violation information.
***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 synthesized 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. At the point of deriving a left-hand side production, the corresponding right-hand side semantic functions are evaluated/executed. Conditions govern the firing of the set of semantic functions directly above them at the point of their satisfaction.

$S \rightarrow K$ $\quad lim(K) = lim(S)$

$K \rightarrow I$ $\quad weight(K) = weight(K) + weight(I)$
**Condition** : $if(usage(K) + weight(I) <= lim(K))$
$items(K) = items(K) + item(I)$

$K_1 \rightarrow IK_2$ $\quad weight(K_1) = weight(K_1) + weight(I)$
$items(K_1) = items(K_1) + item(I)$
$usage(K_1) = usage(K_1) + weight(I)$
**Condition** : $if(usage(K_1) < lim(K_1))$
$lim(K_2) = lim(K_1)$
$usage(K_2) = usage(K_1)$
$items(K_2) = items(K_1)$

$I \rightarrow i_1$ $\quad item(I) =$ "$i_1$"
**Condition** : $if(notinknapsack?(i_1))$

$$weight(I) = weight(i_1)$$

$$\vdots$$

$$I \rightarrow i_n \qquad item(I) = \text{“}i_n\text{”}$$
$$\textbf{Condition}: \quad if(notinknapsack?(i_n))$$
$$weight(I) = weight(i_n)$$

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. Fig. 2 shows the synthesized and inherited message passing in-

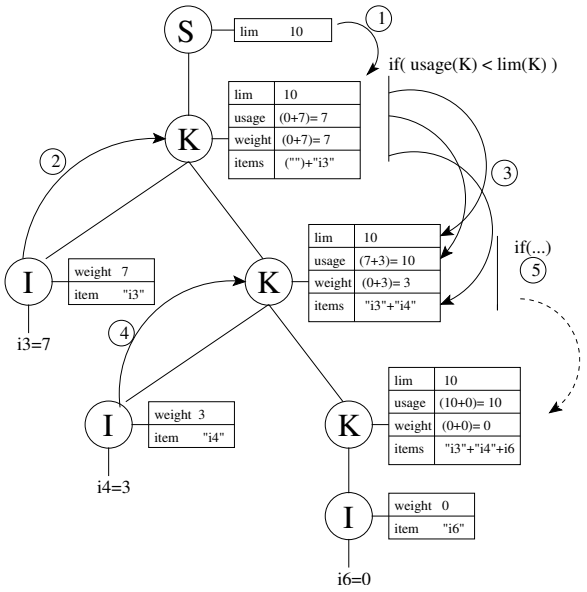


**Fig. 2.** Diagram showing synthesized and inherited message-passing for evaluating attributes in the derivation tree of an attribute grammar.

volved in evaluating derivation trees for the above attribute 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

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

down the tree by the second three semantics functions (otherwise we would have remapped $K$ via another production and repeated the process). The attribute grammar decoder works then by attempting to add items according to $I$'s production rules, and at the point of constraint-violation or a 01 collision the codon causing the error is skipped (becomes an intron) and the subsequent codon read.

## 5 Experimental Setup

In this study, we have chosen to apply our analysis to a range of problem instances from the literature [22], which allow us to gauge the effectivity of different grammars to capture the context-sensitive information for the test-bed knapsack problems. Our current attribute grammar decoder as described in the previous section, uses a simple construction heuristic similar to that of the first-fit heuristic described in [8]. From the literature, this set of problems allow us a direct comparison with two different knapsack problem approaches. Primarily, we undergo a direct comparison with the penalty-based GA of Khuri et al. [13], and a secondary comparison with a hybrid GA which uses a problem-space search [23]. It is worth noting however that Khuri et al. use a direct bitstring representation of fixed-length, where a graded penalty function overcomes the problem of infeasiblity. We utilise the standard variable-length binary string representation of GE, with the attribute grammar mapping as a decoder for feasibility. For experiments with the standard GE mapping process, we penalise to zero - all infeasible candidates. Our earlier work in [24] tested a graded penalty term but it provided no improvement for results.
We adopt standard experimental parameters for GE, changing only the population size to that of Khuri et al., whose a population size of $\mu = 50$ running for up to 4000 generations. We adopt a variable length one-point crossover probability of 0.9, bit mutation probability of 0.01, and roulette 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. The experimental metric of *percentage of runs yielding an optimum solution* serve to demonstrate GE's ability to solve these problems.

## 6 Results

A comparison of the standard GE context-free grammar, the 01 attribute grammar, and the full attribute grammar can be seen in Table 1. The benefit of adopting an attribute grammar on these problem instances are clear with the full constraint checking attribute grammar clearly outperforming the two other grammars analysed. On comparison to the results obtained in [13, 23] it can be seen that the results presented show GE with the attribute grammar decoder

to clearly outperform the traditional GA of Khuri et al. on some instances, and provide competitive results to the hybrid GA of Cotta which uses local optimisation. The results labeled *DE* show the effect of implementing phenotypic duplicate elimination, as described in [6], where we observe that disallowing duplicates at a phenotypic level has the desired effect in improvement of performance. It should be noted, however, that the best of the attribute grammar results fall short of the number of successful solutions found by the best results in the literature.

**Table 1.** Comparing the three grammars, and results from [13, 23] on the percentage of runs achieving an optimum solution. The effect of phenotypic duplicate elimination (DE) is also presented for the full attribute grammar.

| **Problem** | $n$ | $m$ | **GE** | **AG(01)** | **AG(Full)** | **Khuri** | **Cotta** | **AG(Full)+DE** |
|---|---|---|---|---|---|---|---|---|
| knap15 | 15 | 10 | 3.33% | 60% | 83.33% | 83% | 100% | 96.6% |
| knap20 | 20 | 10 | 6.66% | 33.33% | 76.66% | 33% | 94% | 100% |
| knap28 | 28 | 10 | 0% | 3.33% | 40% | 33% | 100% | 90% |
| knap39 | 39 | 5 | 0% | 0% | 36.66% | 4% | 60% | 43.33% |
| knap50 | 50 | 5 | 0% | 0% | 3.33% | 1% | 46% | 16.66% |
| Sento1 | 60 | 30 | 0% | 0% | 10% | 5% | 75% | 66.66% |
| Sento2 | 60 | 300 | 0% | 0% | 3.33% | 2% | 39% | 30% |
| Weing7 | 105 | 2 | 0% | 0% | 0% | 0% | 40% | 0% |
| Weing8 | 105 | 2 | 0% | 0% | 6.66% | 6% | 29% | 36.66% |

## 7 Conclusions & Future Work

We wished to examine the extension of the standard GE mapping process to handle context-sensitive information via the medium of attribute grammars. The results demonstrated a clear advantage for the attribute grammars over the standard context-free grammar on the problem instances examined. Results have also been provided to support the findings of Raidl and Gottlieb [6], which show that duplicate elimination at a phenotypic level can improve performance. More work is required to improve the performance of this approach, and to analyze the redundancy of the attribute grammar decoder in terms of locality and effect of operators (initial results show that this is preserved).

## References


1. Martello, S., Toth, P. (1990). *Knapsack Problems*. J. Wiley & Sons, 1990.
2. Gottlieb, J. (2000). Permutation-Based Evolutionary Algorithms for Multidimensional Knapsack Problem. *Proc. of ACM Symp. on Applied Computing*.
3. Raidl, Gunther R., Gottlieb, J. (1999). Characterizing Locality in Decoder-Based EAs for the Multidimensional Knapsack Problem. *4th European Conference on Artificial Evolution*, pp. 38 - 52, Springer-Verlag.

4. Raidl, Gunther R., Gottlieb, J. (1999). The Effects of Locality on the Dynamics of Decoder-Based Evolutionary Search. *Proc. of the Genetic and Evolutionary Computation Conference*, pp. 787, Morgan Kaufmann.
5. Raidl, Gunther R. (1998). An Improved Genetic Algorithm for the Multiconstrained 0-1 Knapsack Problem. *Proc of 1998 IEEE Congress on Evolutionary Computation*, pp. 207 - 211.
6. Raidl, Gunther R., Gottlieb, J. (1999). On the importance of phenotypic duplicate elimination in decoder-based evolutionary algorithms. *Proc. of the Genetic and Evolutionary Computation Conference*, Late-Breaking Papers, pp. 204-211.
7. Hinterding, R. (1994). Mapping, Order-Independant Genes and the Knapsack Problem. *Proc. 1st IEEE Int. Conf. on Evolutionary Computation*, pp. 13-17 .
8. Hinterding, R. (1999). Representation, Constraint Satisfaction and the Knapsack Problem. *Proc. of 1999 IEEE Congress on EC*, pp. 1286-1292 .
9. Gottlieb J. (1999) Evolutionary Algorithms for Multidimensional Knapsack Problems: the Relevance of the Boundary of the Feasible Region. *Proc. of the Genetic and Evolutionary Computation Conference*, pp. 787, Morgan Kaufman.
10. Gottlieb, J. (1999) On the Effectivity of Evolutionary Algorithms for the Multidimensional Knapsack Problems. *Proc. of Artificial Evolution*, Springer LNCS.
11. Chu, P.C. and Beasley, J.E. (1998). A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics* 4:63-86.
12. Raidl, Gunther R. (1999). Weight-Codings in a Genetic Algorithm for the Multiconstraint Knapsack Problem. *Proc of 1999 IEEE Congress on Evolutionary Computation*, pp. 596-603.
13. Khuri, S., Back, T., and Heitkotter, J. (1994). The zero/one multiple knapsack problem and genetic algorithms. In Deaton, E. et al., editors, *Proc. of the 1994 ACM symposium of Applied Computation,* pp. 188-193, ACM Press.
14. Olsen, A. L. (1994): Penalty Functions and the Knapsack Problems. *in Proc. of the 1st Int. Conf. on Evolutionary Computation*, pp. 559-564.
15. O'Neill, M., Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers.
16. Koza, J.R. (1992). *Genetic Programming*. MIT Press.
17. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann.
18. Knuth, D.E. (1968). Semantics of Context-Free Languages. *Mathematical Systems Theory*, Vol. 2, No. 2. Springer-Verlag.
19. O'Neill, M. (2001). *Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution*. PhD thesis, University of Limerick, 2001.
20. O'Neill, M., Ryan, C. (2001). Grammatical Evolution, *IEEE Trans. Evolutionary Computation*, Vol.5, No.4, 2001.
21. Ryan, C., Collins, J.J., O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Proc. of the First European Workshop on GP*, 83-95, Springer-Verlag.
22. 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.
23. Cotta, C.,Troya, Jose, M (1998). A Hybrid Genetic Algorithm for the 0-1 Multiple Knapsack Problem. *In Artificial Neural Nets and Genetic Algorithms 3*, pp. 251-255, Springer-Verlag.
24. O'Neill, M., Cleary, R., Nikolov, N. (2004). Solving Knapsack Problems with Attribute Grammars. *In Proc. of the Grammatical Evolution Workshop 2004.*