Genetic Programming over Context-Free Languages with Linear Constraints for the Knapsack Problem: First Results

Peter Bruhn Peter.Bruhn@wu-wien.ac.at
Department of Information Business, Vienna University of Economics and Business Administration, Vienna, Austria

Andreas Geyer-Schulz Andreas.Geyer-Schulz@em.uni-karlsruhe.de
Information Services and Electronic Markets, Universität Karlsruhe (TH), Karlsruhe, Germany

Abstract
In this paper, we introduce genetic programming over context-free languages with linear constraints for combinatorial optimization, apply this method to several variants of the multidimensional knapsack problem, and discuss its performance relative to Michalewicz's genetic algorithm with penalty functions. With respect to Michalewicz's approach, we demonstrate that genetic programming over context-free languages with linear constraints improves convergence. A final result is that genetic programming over context-free languages with linear constraints is ideally suited to modeling complementarities between items in a knapsack problem: The more complementarities in the problem, the stronger the performance in comparison to its competitors.

Keywords
Genetic algorithms, grammatical evolution, grammar-based genetic programming, combinatorial optimization, context-free grammars with linear constraints, knapsack problems.

1 Introduction

In this paper, we present genetic programming over context-free languages with linear constraints for combinatorial optimization problems. We perform a first comparison with the penalty function based genetic algorithm of Michalewicz (1994) for variants of the knapsack problem whose multidimensional zero-one version can be formulated as follows:

$$\text{maximize} \quad \mathbf{p}^T\mathbf{x} \tag{1}$$
$$\text{subject to} \quad \mathbf{W}\mathbf{x} \leq \mathbf{b} \tag{2}$$
$$x_j \in N_0 \quad \forall j, \tag{3}$$
$$\mathbf{b} > \mathbf{0}, \mathbf{p} > \mathbf{0}, w_{ij} > 0 \quad \forall i, j, \tag{4}$$

where $\mathbf{x}$ represents the n-dimensional vector of items in the knapsack, $\mathbf{p}$ the n-dimensional vector of prices of the items, $\mathbf{b}$ the m-dimensional vector of constraint bounds for each of the m constraints, $\mathbf{W}$ the $m \times n$-dimensional coefficient matrix of the constraints, and $i = 1 \ldots m$, $j = 1 \ldots n$. Note that we get the multidimensional zero-one version of the knapsack problem by replacing constraint (3) with $x_j \in \{0, 1\} \quad \forall j$.

Why is the problem still of considerable interest? First, the multidimensional zero-one knapsack is a well-known, NP-hard problem, and it is equivalent to any zero-one integer programming problem with non-negative coefficients, a view taken in most early work in this area (e.g., Hillier (1969)). Second, many practical problems can be solved as a knapsack problem, e.g., cutting stock problems (Gilmore and Gomory, 1966), resource allocation in distributed computer systems (Gavish and Pirkul, 1982), container/vehicle loading (Shih, 1979), and more recently, combinatorial auctions (Rothkopf et al., 1998). Third, variants with varying degrees of difficulty and a set of benchmark problems available from the OR-Library (Beasley, 1990, 1996) make the knapsack problem and its variants attractive as a testbed for new methods.

For a review of the literature on the knapsack problem, we refer the reader to Chu and Beasley (1998). We restrict our presentation in the following to a short discussion of the main ideas, how genetic algorithms can be extended for combinatorial optimization problems, in principle, and how suitable and successful these ideas are when applied in the knapsack problem domain. For clarity of exposition, we distinguish the following three strategies; each allows for a number of refinements and can be mixed together:

1. Infeasible solution candidates.
2. Hybrid genetic algorithms.
3. Feasible solution candidates.

1.1 Infeasible Solution Candidates

This strategy allows for the generation of infeasible solution candidates that violate some of the constraints of the combinatorial optimization problem. We deal with the violation of constraints either with penalty functions or with repair or decoder algorithms. Penalty functions reduce the fitness of a solution candidate that violates a constraint. The problem of calibrating these functions, raised in Goldberg (1987), is discussed in the context of a small set-covering problem in Richardson et al. (1989). The genetic algorithm of Michalewicz (1994), which we show in more detail in Section 3.1, represents the penalty function approach to the knapsack problem as does the algorithm of Khuri et al. (1994) who report only moderate results on a small number of standard problems. However, if additional nonlinear constraints must be respected in a knapsack problem, penalty functions are a natural choice. For example, see Powell and Skolnick (1993).

The basic idea of repair and decoder algorithms is to find an algorithm that is guaranteed to translate an infeasible solution candidate into a feasible one. The difference between a repair algorithm and a decoder is that in a repair algorithm, the infeasible solution candidate is replaced by a feasible one in the population of a genetic algorithm, while a decoder just computes the fitness of the infeasible solution candidate from the decoded feasible solution candidate. A decoder leaves the infeasible solution candidate in the population of the genetic algorithm. The algorithm of Chu and Beasley (1998) for the multidimensional knapsack problem is distinguished by its repair algorithm explained in the next paragraph. Note that a repair algorithm often requires a problem-specific heuristic and is one of the spots for hybridization, that is, integrating a competing algorithm. A comparison of several applications of decoder-based evolutionary algorithms for the multidimensional zero-one knapsack problem is given in Gottlieb and Raidl (2000).

1.2 Hybrid Genetic Algorithms

Hybrid genetic algorithms are genetic algorithms that integrate competing algorithms or heuristics. For an introduction to hybrid genetic algorithms, see Davis (1991). Chu and Beasley (1998) integrate the pseudo utility ratio $\mu_j = p_j \sum_{i=1}^{m} \omega_i \mathbf{W}_{ij}$ into their greedy repair algorithm, where the ω_i are the values of the dual variables (the shadow prices) of the ith constraint of the linear programming relaxation of the multidimensional knapsack problem. Raidl (1998) improves Chu and Beasley's algorithm, in principle, with a randomized repair strategy based on the values of the primal variables of the relaxed linear programming solution. Raidl (1999) investigates weight coding combined with several relaxation-based decoding heuristics.

Other attempts at hybridization of genetic algorithms are reported in Thiel and Voss (1994) who experiment with several local search algorithms and recommend a tabu search heuristic. Rudolph and Sprave (1996) suggest the following modifications: selection is restricted to neighboring solutions, infeasible solutions are penalized, and an adaptive threshold acceptance schedule is "borrowed" from Dueck (1993).

1.3 Feasible Solution Candidates

This strategy concentrates on generating only feasible solution candidates by an appropriate choice of representation with suitable genetic operators that guarantee constraints are never violated. An example of this approach is the genetic algorithm of Hoff et al. (1996), which achieves this with the help of a random repair algorithm. To implement this strategy, we can distinguish two different approaches: test and reject methods, and problem representations and operators with guaranteed feasible solutions. In test and reject methods, solution candidates are randomly generated/modified, and infeasible solution candidates are rejected. This approach is often used in the design of random number generators for multivariate distributions (Robert, 1999, chapter 2.3). However, no evolutionary algorithms of this type are known to us.

An example of the second approach is the development of order-based genetic algorithms for the traveling salesman problem and other variants of scheduling problems. The basic characteristic of this class of algorithms is that a solution candidate for a scheduling problem can be represented as a sequence of alleles, and suitable genetic operators can be defined in terms of exchanging two arbitrarily selected alleles so that the resulting sequence remains feasible. A comparison of such sequencing operations can be found in Starkweather et al. (1991). Note that although this approach does not seem to be suitable for the multidimensional knapsack problem, for combinatorial optimization problems like the vehicle routing problem, which combine scheduling and knapsack type problems, this approach might provide a point of departure for modeling the sequencing aspects of the problem.

Another variant of the second approach is the following: Instead of representing a knapsack as a bit-string whose bits set indicate the items in the knapsack, one may use, for example, a robot language that describes the program of a robot that actually packs items in a knapsack until a solution candidate is found. Such a program can be evolved by a genetic program for solving this type of optimization problem. In this approach, the constraints must be integrated in the robot simulation environment. The (optimal) genetic program is a description of the production process with which a solution can actually be produced. This idea was suggested under the heading *emergent intelligence* by Angeline (1994). For a certain type of cutting-stock problem, Auer (1996) developed a programmable simulator of a production line, a machine programming language for the simulator, and a specialized integer-coded genetic algorithm for breeding machine programs, which solves the cutting-stock problem. In a series of experiments, he compared his approach to exact solutions found using the method of Gilmore and Gomory (1966) and reported favorable results for low-volume order sets with a high number of different cutting patterns.

In our work, we use a representation based on a language that allows only feasible solutions. Other work in the literature has achieved the same objective. Strongly-typed genetic programming adds constraints to genetic programming with the help of a type system (Montana, 1995). Another approach is grammar-based genetic programming. We attribute to Antonisse (1991) the concept of grammar-based genetic algorithms for general languages, including type-0 (unrestricted) and type-1 (context-sensitive), and the development of a syntactically closed crossover operator. The part of the theory for genetic algorithms over type-2 (context-free) languages was developed for the comparison of crisp- and fuzzy-rule languages in Geyer-Schulz (1991) (a full version of this appeared as Geyer-Schulz (1993)). Algorithms in APL2 were implemented in 1993 and published in Geyer-Schulz (1994), where explicit representations of grammars were used in two applications: evolving 3-D models of jet-planes with a hybrid GA/GP combination (Nguyen and Huang, 1994), and genetic microprogramming of neural networks (Gruau, 1994). The first extension of simple genetic algorithms to context-free languages and their analysis with the help of formal power series is in Geyer-Schulz (1994), which finally appeared as Geyer-Schulz (1995) and revised as Geyer-Schulz (1996). A C++ implementation more or less based on the APL2 implementation in Geyer-Schulz (1995, chapter 8) has been developed by Hörner (1996a, 1996b). The same class of algorithms has been independently developed by Whigham (1995a, 1995b, 1995c, 1996a, 1996b), albeit without analysis by formal power series. The algorithms of Whigham and Geyer-Schulz are both based on derivation trees. Gruau (1996) adopts a similar approach for improving genetic programming performance. In a logic programming context, Wong and Leung (1996) have developed a genetic algorithm based on definite clause grammars, which evolved from Wong and Leung (1995). As Keller and Banzhaf (1996) demonstrated, grammar-based genetic programming is not necessarily linked to a tree representation. Programs in arbitrary context-free languages can be generated by a clever interpretation of a set of random numbers as choices of productions of a grammar (and a suitably defined set of default rules for tree completion). Some work on grammatical evolution is based on the same idea (Ryan et al, 1998; Ryan and O'Neill, 1998; O'Neill, 1999; O'Neill and Ryan, 1999a, 1999b, 1999c, 1999d, 2001). Note that in these approaches, the process of generating a program from a grammar is coded. This is very similar to the idea of Angeline outlined above.

With regard to multidimensional knapsack problems, all of these genetic programming methods share the same deficiency: the constraint system of the problem cannot be represented directly by the language describing the knapsack. No extension of genetic programming to combinatorial optimization problems is known to us. But as we show in Section 3.1, the integer-valued multidimensional knapsack problem can be represented by a context-free language **and** a set of linear constraints.

1.4 Outline

In Section 2, we present genetic programming over context-free languages with linear constraints, which extends the work of Geyer-Schulz (1996) with a constraint propagation mechanism for linear constraints. This new algorithm is a pure representative of the strategy to allow only feasible solution candidates and the first extension of genetic

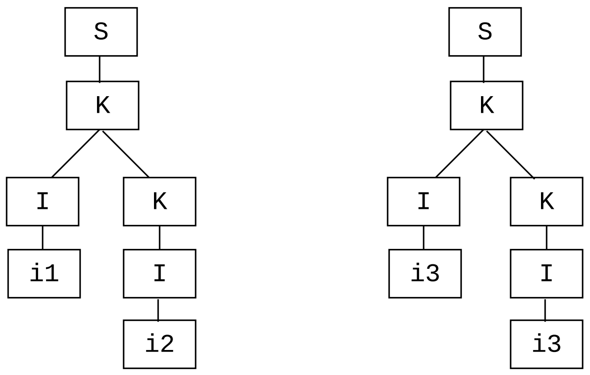

Figure 1: Two derivation trees.

programming for combinatorial optimization problems in this way. A full account of this work including implementation details and source code is given by Bruhn (2000).

In Section 3, we investigate the performance of this algorithm for an integer-valued multidimensional knapsack problem compared to the genetic algorithm of Michalewicz (Section 3.1) and for an integer-valued multidimensional knapsack problem for complementary goods, again compared with the genetic algorithm of Michalewicz (Section 3.2).

In Section 4, we discuss the limitations and problems of our new algorithm and identify directions for research and improvement of the algorithm.

2 Genetic Programming over Context-Free Languages with Linear Constraints

2.1 A Motivating Example

As an illustration of the idea of genetic programming over context-free languages with linear constraints, consider the following knapsack problem:

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 5x_2 + 3x_3 && (5)\\ \text{subject to} \quad & 4x_1 + x_2 + 5x_3 \leq 9 && (6)\\ \text{with} \quad & x_1, x_2, x_3 \in N_0 && (7) \end{aligned}$$

The following context-free grammar with start symbol S represents all knapsacks of this problem, with x_1, x_2, and x_3 denoting the number of items i_1, i_2, and i_3 contained in the knapsack:

$$\begin{aligned} S &\rightarrow K, && (8)\\ K &\rightarrow I|IK, && (9)\\ I &\rightarrow i_1|i_2|i_3, && (10) \end{aligned}$$

where each terminal symbol i_1, i_2, and i_3 denotes an item that could be contained in the knapsack. For example, i_1 may represent a battery pack, i_2 a flash light, and i_3 a cellular phone.

Figure 1 shows two derivation trees for this grammar, the left tree represents the knapsack $i_1 i_2$ with a value of 7 and a weight of $4 + 1 = 5$, which is below the limit of 9.

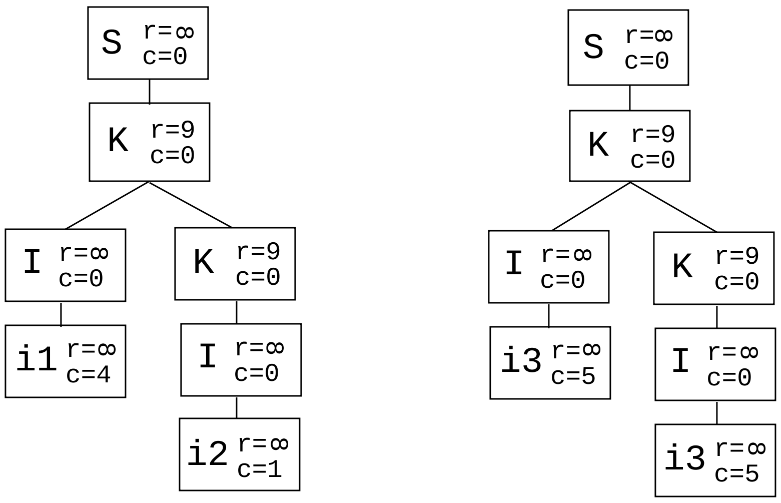

Figure 2: Two derivation trees with linear constraints.

The right tree represents the knapsack $i_3 i_3$ with a value of 6 and a weight of $5 + 5 = 10$, which is infeasible because its weight is above the limit of 9. Obviously, by adding a constraint check that discards infeasible derivation trees to the grammar-based genetic programming algorithm of Geyer-Schulz (1996), we can immediately solve our example problem at the cost of generating and then discarding infeasible derivation trees. Unfortunately, the cost of testing and rejecting derivation trees increases with problem size.

We can try to keep track of constraint (6) by extending the information stored in a node of the derivation tree in a suitable manner that supports testing the feasibility of a derivation tree and generating and substituting feasible derivation trees.

As a first step in this direction, we have to represent the constraint

$$4x_1 + x_2 + 5x_3 \leq 9 \tag{11}$$

in the derivation trees shown in Figure 1. We note that the terminal symbols i_1, i_2, and i_3 have weights of 4, 1, and 5, respectively. A knapsack is denoted by the nonterminal symbol K, and its weight is restricted to 9. In Figure 2, we integrated these coefficients into the derivation trees shown in Figure 1. Note that only the nonterminal symbol K has a restriction $r = 9$, whereas all other symbols are unrestricted ($r = \infty$). Only the terminal symbols i_1, i_2, and i_3 have weights $c > 0$; all nonterminals have a weight of 0 ($c = 0$).

In Figure 2, we achieved an integration of the linear constraint (6) into the derivation tree in a systematic way. But what we really want (and need) to know is:

1. How much weight has been used up by each derivation subtree (usage u)?
2. What is the weight limit for replacing a derivation subtree with another derivation subtree with the same root (limit l)?

As Figure 3 shows, we recursively compute the usage of a node in the derivation tree as the sum of its own usage and the usages of its sons as defined in Equation (12).

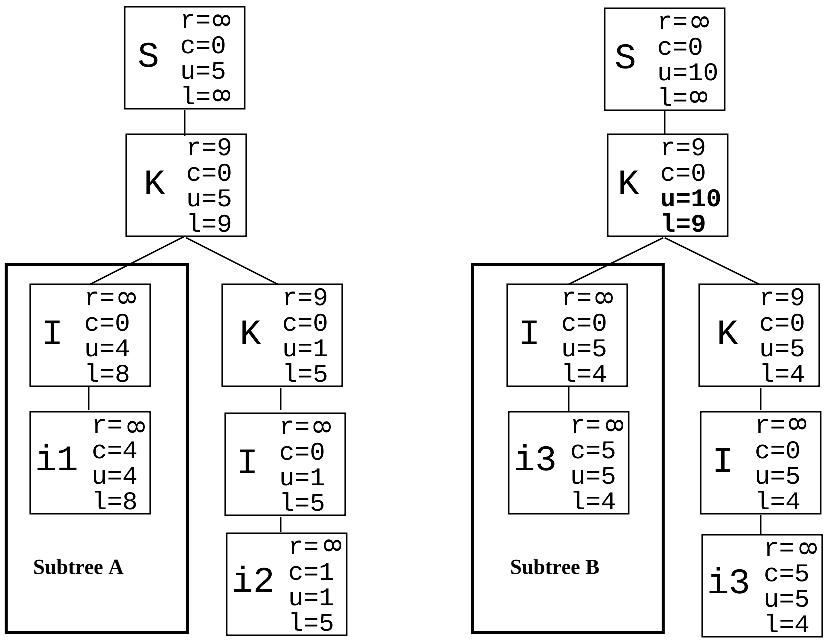

Figure 3: Two derivation trees with linear constraints, usages, and limits.

For example, consider node K with $c = 0$, sons I, $u = 4$, and K with $u = 1$ in the left derivation tree of Figure 3. We get $u_K = 5 = 0 + 4 + 1$.

The limit of a node is the minimum of its own restriction r and the sum of its own usage added to the difference of the limit of its father and the usage of its father as defined in Equation (13). Consider node I with $r = \infty$ and $u = 4$ and its father node K with $l = 9$ and $u = 5$ in the left derivation tree of Figure 3. We get $l_I = 8 = \min(\infty, 9 - 5 + 4)$. That is, resource usage by other sons has to be taken into account. Clearly, a derivation tree does not violate constraint (6) if $l \geq u$ holds in all of its nodes. We immediately see that the right derivation in Figure 3 violates constraint (6), because in the second node K, we see that $l = 9$ and $u = 10$ (bold in Figure 3).

We see that we can substitute the subtree A for the subtree B in Figure 3, resulting in the derivation tree shown in Figure 4, where the recomputed values are in bold.

2.2 Context-Free Grammars with Linear Constraints

We start with a repetition of a few basic definitions for formal languages (Hopcroft and Ullman, 1979):

An *alphabet* is a finite set of symbols. A *string* is a finite sequence of symbols. A (formal) language Σ is a (finite or infinite) set of strings based on a chosen alphabet. We denote the set of all strings over a given alphabet by Σ^*, the set of all strings with length n over a given alphabet by Σ^n.

A *context-free grammar* is a tuple $G = (V, T, P, S)$, where V is the set of nonterminal symbols, T the set of terminal symbols with $V \cap T = \emptyset$, the relation $P \subseteq V \times (V \cup T)^*$ is a finite set of production rules and S the start symbol with $S \in V$. We denote productions by $A \rightarrow \alpha$ (A derives α), where $A \in V$ is a non-terminal and $\alpha \in (V \cup T)^*$ a string. The set of all strings consisting only of terminals that can be derived from start symbol

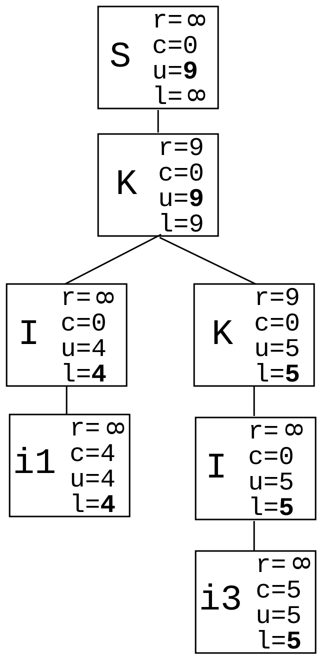

Figure 4: Derivation tree after substitution.

S by sequential application of production rules is called the *context-free language* $L(G)$, which is produced from grammar G. (The derivation $A \to \alpha$ can be seen as substitution $\sigma(A, \alpha)$, too.)

Every string of a context-free language with a context-free grammar $G = (V, T, P, S)$ can be represented as a *derivation tree* with the following properties:

1. The root is marked with S.
2. Every inner node is marked with a symbol from V.
3. Every leaf node is marked with a symbol from T.
4. If node n is marked with A and the sons of node n from left to right are marked with $X_1, X_2, \ldots, X_k$, then there must be a production rule $A \to X_1 X_2 \ldots X_k$ in P.

For the purpose of grammar-based genetic programming, we note that a grammar can be used as a device for generating random strings of a context-free language by randomly selecting which derivation should be done next. A formal analysis of such devices can be found in Böhm and Geyer-Schulz (1997).

In addition, the building block for genetic operators (like mutation and crossover) in grammar-based genetic programming is feasible subtree substitution that is closed over $L(G)$, as shown in Geyer-Schulz (1996): Suppose D is a complete derivation tree for a context-free grammar $G = (V, T, P, X)$ with a set of subtrees $D_1, \ldots, D_n$ whose roots are labeled with symbols from $V \cup T$. I is a complete derivation tree for the context-free grammar $G' = (V, T, P, S)$ whose root is labeled with S. A subtree substitution is feasible, if I replaces a subtree D_i of D, which is labeled with S (Geyer-Schulz, 1996). To guarantee that feasible subtree substitution is closed, we complete it with an identity operation: if no subtree labeled with S exists in D, the result of the operation is D.

In the context of combinatorial optimization problems, we can interpret each string of a suitably defined, context-free language as a solution candidate for the optimization problem. Unfortunately, a context-free grammar allows for the generation of strings that violate one or more of the constraints that are part of the combinatorial optimization problem. Since in this paper we concentrate on the knapsack problem and its variants, it suffices to integrate linear constraints of the form $\mathbf{Wx} \leq \mathbf{b}$ with context-free grammars for grammar-based genetic programming.

DEFINITION 1 (Context-Free Grammar with Linear Constraints): *A set of linear constraints* $\mathbf{Wx} \leq \mathbf{b}$, $\mathbf{W} = [\mathbf{w}_1|\ldots|\mathbf{w}_n]$ *(with* $x_j \in N_0\ \forall j, \mathbf{b} > \mathbf{0}, w_{ij} > 0\ \forall i,j,$ *with* $i = 1,\ldots,m, j = 1,\ldots,n$*) is integrated with a context-free grammar* $G = (V,T,P,S)$ *in the following way:*

1. *Each symbol in* $V \cup T$ *is represented by a triple* $(e, \mathbf{r}, \mathbf{c})$*, where* e *denotes the label of the symbol,* $\mathbf{r}$ *the* m*-dimensional vector of bounds (*$r_i \in R \cup \infty$*), and* $\mathbf{c}$ *the* m*-dimensional vector of technical coefficients of* e*.*

2. *For the start symbol* S*:* $\mathbf{r} = \infty$*, and* $\mathbf{c} = \mathbf{0}$*.*

3. *For the special non-terminal symbol, say* K*, that represents the configuration of items (e.g., the knapsack):* $\mathbf{r} = \mathbf{b}$*, and* $\mathbf{c} = \mathbf{0}$*.*

4. *For all other non-terminal symbols:* $\mathbf{r} = \infty$*, and* $\mathbf{c} = \mathbf{0}$*.*

5. *For the terminal symbol representing* i_j*, counted by variable* x_j*:* $\mathbf{r} = \infty$*, and* $\mathbf{c} = \mathbf{w}_j\ \forall j$*.*

In the examples in Section 3, we denote the assignment of vectors $\mathbf{r}$ and $\mathbf{c}$ to the symbol s of $V \cup T$ as $\mathbf{r}(s)$ and $\mathbf{c}(s)$ as suggested in Bruhn (2000). Note that we can use a context-free grammar with linear constraints as a word-generating device in the same way as a context-free grammar.

In the derivation trees generated from such a context-free grammar with linear constraints, we keep track of the constraints in each node e of the derivation tree with a vector $\mathbf{u}_e$ of resource usage and $\mathbf{l}_e$ of limits available for the subtree labeled e given the rest of the derivation tree.

We recursively define the usage of resources $\mathbf{u}_e$ for each node e in a derivation tree by computing

$$\mathbf{u}_e = \mathbf{c}_e + \sum_{v \in S(e)} \mathbf{u}_v, \tag{12}$$

where $S(e)$ is the set of the sons of node e.

The limit $\mathbf{l}_e$ of a node e is the amount of resources the subtree maximally could have used. Once again, we compute $\mathbf{l}_e$ for all nodes of a derivation tree (except the root) recursively:

$$\mathbf{l}_e = \min[\mathbf{r}_e, (\mathbf{l}_{V(e)} - \mathbf{u}_{V(e)} + \mathbf{u}_e)], \tag{13}$$

where $V(e)$ is the father of node e.

A derivation tree for a context-free grammar with linear constraints does not violate constraints $\mathbf{Wx} \leq \mathbf{b}$, if $\mathbf{1} \equiv (\mathbf{l}_e \geq \mathbf{u}_e)\ \forall e$ with $\mathbf{x} \equiv \mathbf{y}$ defined as 1 if $x_i = y_i\ \forall i = 1,\ldots,m$ and $\mathbf{x}$ and $\mathbf{y}$ m-dimensional vectors, and $\mathbf{x} \equiv \mathbf{y}$ defined as 0 otherwise. When suitable, 0 stands for *false* and 1 stands for *true*. Finally, we extend

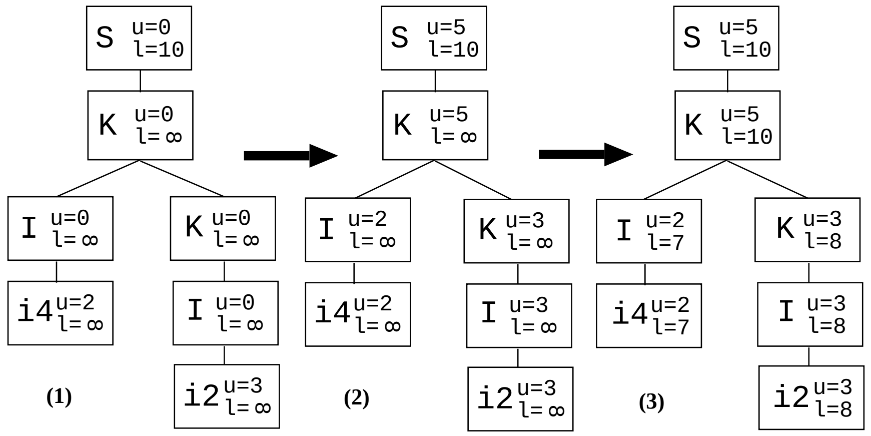

Figure 5: Generating an individual.

the definition of a feasible subtree substitution for derivation trees of context-free grammars with linear constraints as follows:

DEFINITION 2 (Feasible Subtree Substitution with Linear Constraints): *A feasible subtree D of a derivation tree of a context-free grammar with linear constraints may replace a subtree E of a derivation tree of the same grammar if*

1. *the root nodes $W(D)$ of D and $W(E)$ of E are labeled with the same symbol,*
2. $\mathbf{1} \equiv (\mathbf{u}(W(D)) \leq \mathbf{l}(W(E)))$ *holds. (Usage of resources is less than resources available.)*

   And if these conditions do not hold, we complete with an identity operation.

2.3 The Genetic Programming Algorithm for Context-Free Grammars with Linear Constraints

In this section, we describe the algorithm used in the experiments in Section 3. The algorithm has been implemented in OCAML.[1] We use a generation-based genetic algorithm with elitism.

Selection is proportional to fitness and based on Baker's (1987) stochastic universal sampling algorithm. It is modified for elitism so that the best individual always remains in the population.

Conceptionally, generating an individual is a process with three phases:

1. In the first phase, we generate a random derivation tree from a context-free grammar with linear constraints by randomly choosing a production rule from the set of production rules suitable for the expansion of a non-terminal symbol. Derivation tree (1) in Figure 5 is the result of the first phase.

2. In the second phase, we compute the usage of resources of each node of the derivation tree as defined in Equation (12). Derivation tree (2) in Figure 5 illustrates the result of this phase.

[1]See `http://caml.inria.fr/ocaml`; its source is contained in Bruhn (2000) and can be obtained by request from the authors.

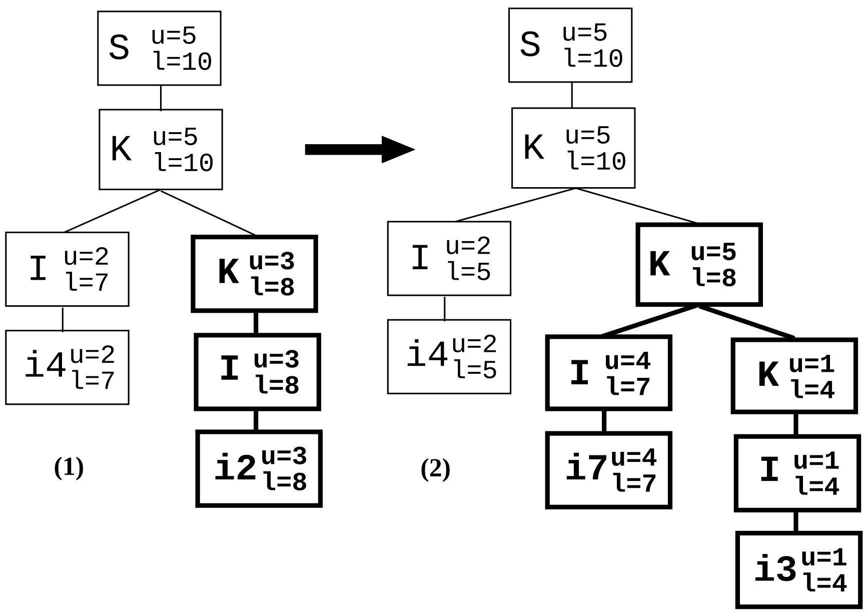

Figure 6: Mutation.

3. In the third phase, we apply Equation (13) recursively to compute the limits of the nodes of the derivation tree. Derivation tree (3) in Figure 5 shows the result of this phase.

In the current implementation, the random generation of a derivation tree is repeated until a feasible individual has been generated.

This implementation of the generation of a derivation tree for a context-free grammar is assumed to work reasonably well as long as the linear constraints are not restricting the search space in such a way that only a few or none of the words are feasible solutions. In this case, however, the generation of an initial population may need quite some time, because of the high number of retries due to constraint violations. In the current version, this drawback can be reduced by reducing the number of derivation steps allowed in the generation process or specifying redundant, additional production rules that favor the derivation of smaller trees. Allowing for redundant production rules in a context-free grammar is a crude but readily available method of changing the search behavior of the genetic programming algorithm.

Mutation is defined as feasible subtree substitution with linear constraints of a randomly selected subtree by a new, randomly generated subtree such that Definition 2 holds. The first step consists of randomly selecting an interior node of the derivation tree with equal probability, which must be labeled by a non-terminal symbol and regenerating this subtree. This must be possible because at least one such subtree exists, namely the subtree that should be replaced. Finally, all l- and u-values of the tree must be recomputed. Figure 6 illustrates the mutation operator. The bold subtree in derivation tree (1) is replaced by the bold subtree in derivation tree (2). In the current version, the mutation rate specifies the probability that a single mutation operation is applied to an individual (a "one-subtree" mutation).

Recombination of two individuals is based on feasible subtree substitution with

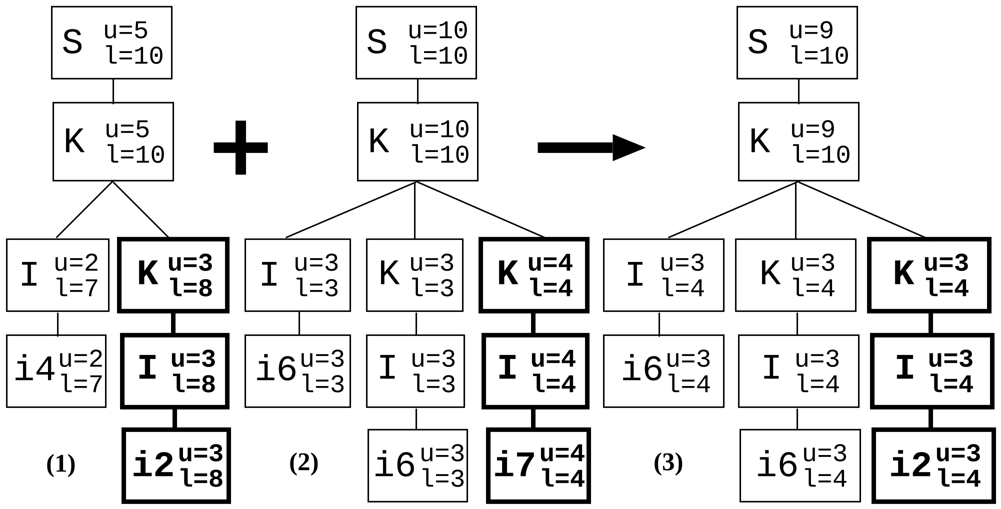

Figure 7: Recombination.

linear constraints given in Definition 2. A randomly selected subtree of the first individual is substituted by a suitable subtree of the second individual such that Definition 2 holds. In detail, for each interior node of individual 1, we generate a list of feasible subtree substitutions, and we choose from all these substitutions with equal probability. An example of a feasible recombination is shown in Figure 7. The bold subtree in derivation tree (1) replaces the bold subtree in derivation tree (2), and the result is derivation tree (3) in Figure 7. Obviously, after the subtree substitution, the l- and u-values of the resulting subtree must be recomputed.

In the current version, two parents always produce only one child for recombination. This is in contrast to other genetic programming variants that implement recombination by exchanging subtrees. The rationale for this change is that with an additional linear constraint system, it may be possible that only one of the two subtree substitutions is feasible and allowing only recombinations where both subtree substitutions are feasible would severely restrict the set of possible recombinations of two individuals.

3 First Results for Variants of the Knapsack Problem

In the following, we present first performance results of genetic programming over context-free languages with linear constraints for:

1. An integer-valued, multidimensional knapsack problem compared to the penalty function based genetic algorithm of Michalewicz (1994).

2. An integer-valued, multidimensional knapsack problem for complementary goods compared with the same algorithm.

In the description of these two experiments, we provide a short problem description and a characterization of the data set for the experiment; we describe the competing algorithms and their parameters for the experiment; and we describe the experiment itself and the inferences we can draw from it.

Table 1: Parameters of experiments 1, 2a, and 2b.

	GA	GP
P(mutation)	0.05	0.05
P(crossover)	0.70	0.70
Selection	SUS, elitist	SUS, elitist
Population	300	300
Generations	300	300
Runs	500	500

3.1 The Integer-Valued Multidimensional Knapsack Problem

3.1.1 Problem

In the integer-valued multidimensional knapsack problem, we maximize the value of items that we pack into a knapsack subject to several linear constraints on, e.g., the weight of the knapsack, the volume of the knapsack, and the assumption that several items of a kind are available. For this problem, we refer the reader to Equations (1)–(4) in Section 1.

3.1.2 Test Data Set

The test data set chosen for this experiment is the first data set of file `mknapcb7.txt`.[2] The method for generating this data set is described in Chu and Beasley (1998). The data set contains a knapsack problem with 100 different types of items subject to 30 constraints; its optimal solution is unknown ($n = 100$, $m = 30$). This data set was originally constructed as a test for zero-one knapsack algorithms.

3.1.3 The Genetic Algorithm of Michalewicz (GA)

Following Michalewicz (1994), we propose the following fitness function $f(\mathbf{x})$ with penalty term $P(\mathbf{x})$ for genetic algorithms subject to linear constraints:

$$f(\mathbf{x}) = \mathbf{p}^T\mathbf{x} - P(\mathbf{x}) \tag{14}$$

$$\text{where} \quad P(\mathbf{x}) = \rho(\mathbf{v} > \mathbf{0})^T\mathbf{v} \tag{15}$$

$$\text{and} \quad \mathbf{v} = (\mathbf{W}\mathbf{x} - \mathbf{b}) \tag{16}$$

$$\rho = \max_{i=1}^{m} \max_{j=1}^{n} (\tfrac{\mathbf{e}\mathbf{p}^T}{\mathbf{W}})_{i,j}. \tag{17}$$

with the m-dimensional column vector $\mathbf{e} = (1, \dots, 1)^T$ and $\frac{\mathbf{A}}{\mathbf{B}}$ as elementwise division of corresponding matrix elements $\frac{a_{ij}}{b_{ij}}$ for two matrices $\mathbf{A}$, $\mathbf{B}$ of the same dimensions. For convenience, $\mathbf{a} > \mathbf{b}$ is defined as elementwise comparison of corresponding vector elements.

A knapsack is coded as a variable length vector of integers from 1 to n, where each integer j represents an item of type j (e.g., the solution vector $(10, 5, 3, 5, 5)$ corresponds to the knapsack solution $x_3 = 1, x_5 = 3, x_{10} = 1$, and $x_j = 0$ for $j \neq 3, 5, 10$).

The initial population consists of vectors of integers from 1 to 100 drawn with equal probability. The initial length of each vector is an integer drawn with equal probability from 1 to 7.

[2]This can be obtained from `ftp://mscmga.ms.ic.ac.uk/pub/mknapcb7.txt` of the OR-Library (`http://www.ms.ic.ac.uk/info.html`).

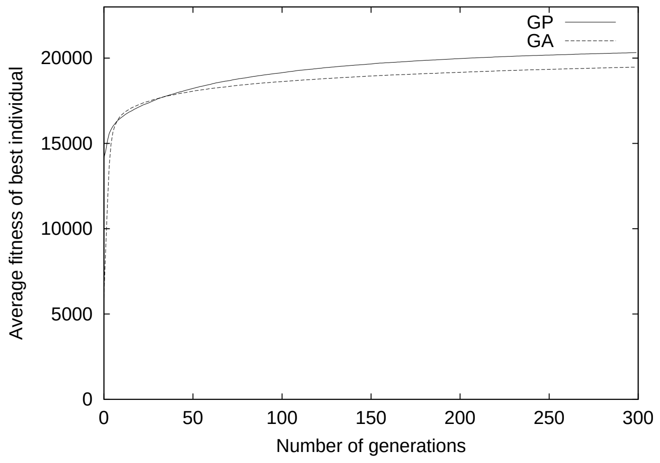

Figure 8: Comparison of GA and GP average performance in experiment 1 (500 runs).

Mutation is applied elementwise; crossover is a one-point crossover operator adapted for variable length strings: crossover positions are selected independently for each parent vector. Selection is done by stochastic universal sampling (SUS), selection is elitist. The parameters used in the experiment are shown in Table 1.

In all runs in all experiments (1500 runs), the best individual of this genetic algorithm was feasible and, therefore, without penalty term.

3.1.4 Genetic Programming with Linear Constraints (GP)

For the genetic programming algorithm of Section 2, the following context-free grammar G_1 with linear constraints is used in the experiment:

$$S \rightarrow K, \tag{18}$$
$$K \rightarrow A|AK|AKK, \tag{19}$$
$$A \rightarrow a_1|a_2|\ldots|a_{100}. \tag{20}$$

The terminal symbols a_j represent items of type j with $j = 1, \ldots, 100$. Note that the production AKK is redundant—its only purpose is to change the probability distribution of derivation trees. It increases the probability of generating knapsacks that contain "more" items.

In addition, the linear constraint set $\mathbf{W}\mathbf{x} \leq \mathbf{b}$ leads to the following allocation of 2 m-dimensional vectors $\mathbf{c}(s)$ and $\mathbf{r}(s)$ to each symbol $s \in (V \cup T)$:

$$\mathbf{c}(a_j) = \mathbf{w}_j \quad \forall j,\ \mathbf{c}(S) = \mathbf{c}(A) = \mathbf{c}(K) = \mathbf{0}, \tag{21}$$
$$\mathbf{r}(S) = \mathbf{b},\ \mathbf{r}(K) = \mathbf{r}(A) = \mathbf{r}(a_j) = \infty \quad \forall j, \tag{22}$$

where $\mathbf{W} = (\mathbf{w}_1, \ldots, \mathbf{w}_n)$ is partitioned in column vectors $\mathbf{w}_j$. Again, the parameters of the genetic programming algorithm used in the experiment are presented in Table 1.

3.1.5 Experiment 1

To compare the average performance of GP with respect to GA, we ran both algorithms 500 times with the parameter settings shown in Table 1.

Table 2: Test for H_0^k versus H_A^k in experiment 1.

k	50	100	150	200	250	300
$\bar{x}_{GP,k}$	18192.17	19128.34	19649.96	19967.34	20175.37	20324.17
$\bar{x}_{GA,k}$	18043.85	18613.91	18946.10	19163.66	19334.93	19472.21
$s^2_{GP,k}$	282520.38	233186.13	207547.33	196776.39	172135.19	170300.06
$s^2_{GA,k}$	176896.31	191146.73	176846.42	169702.02	179705.77	173959.08
Z_k	4.89	17.66	25.39	29.69	31.68	32.47

Figure 8 shows the average fitness of the best individual for the two algorithms, GA and GP, for each generation . A first visual inspection of Figure 8 shows the following:

1. The expected average fitness of the best individual in a GP start population is far higher than in the GA start population (14147 for GP versus 5951 for the GA). The best individual in all GA start populations is feasible. This implies that the penalty function does not contribute to the inferior start solutions in the GA.

2. GP seems to produce better solutions in the long run.

To test the impression that the average performance of the GP is better than GA, we set up the following test for each generation: Let $\bar{x}_{GP,k}$, $\bar{x}_{GA,k}$, $s^2_{GP,k}$, and $s^2_{GA,k}$ denote the sample means and variances of the fitness of the best individual at generation k for a sample of 500 independent runs of GP and GA. Let $\mu_{GP,k}$, $\mu_{GA,k}$, $\sigma^2_{GP,k}$, and $\sigma^2_{GA,k}$ denote the true means and variances. Since the observed sample values $x_{GP,k}$ and $x_{GA,k}$ are independently identically distributed (i.i.d.) by the central limit theorem, we expect $\bar{x}_{GP,k}$ and $\bar{x}_{GA,k}$ to be approximately normally distributed with means $\mu_{GP,k}$ and $\mu_{GA,k}$ and variances $\sigma^2_{GP,k}$ and $\sigma^2_{GA,k}$. For testing

$$H_0^k : \mu_{GP,k} - \mu_{GA,k} = 0 \tag{23}$$

with respect to the alternative hypothesis

$$H_A^k : \mu_{GP,k} > \mu_{GA,k} \tag{24}$$

we use the standard test statistic

$$Z_k = \frac{\bar{x}_{GP,k} - \bar{x}_{GA,k}}{\sqrt{\frac{s^2_{GP,k}}{n_{GP}} + \frac{s^2_{GA,k}}{n_{GA}}}}, \tag{25}$$

which is asymptotically standardnormal under the null-hypothesis, where n_{GP} and n_{GA} represent the number of runs of the algorithms GP and GA.

Setting the probability of a type I error α to 0.05, we have to dismiss the null-hypothesis if $Z_i > 1.65$.

Table 2 shows these statistics for several generations. We have to reject H_0^k for all cases shown in the table. Tests for all generations have rejected H_0 for all generations but 8 to 39. We conclude that GP outperforms GA during the first 7 generations and from generation 40 to 300. GP seems to be a good anytime algorithm.

3.2 The Integer-Valued Multidimensional Knapsack Problem for Systems (Complementary Goods)

3.2.1 Problem

In this problem, we maximize the value of systems we pack into a knapsack. The knapsack is subject to several linear constraints and the system constraints. The system constraints enforce that each system is fully composed of components of proper type. In economics, complementary effects between goods are well known, and the impact of systems consisting of a number of compatible and complementary goods on competition and standardization has been analyzed in Katz and Shapiro (1985). A practical example is the assembly of PCs. For experiments 2a and 2b, we restrict ourselves to the simplest version of this problem—systems consisting of two types of components, C and D, each having a number of compatible items available, and the assumption that the value of a system is simply the sum of the value of its components:

$$\begin{aligned} \text{maximize} \quad & \mathbf{p}^T\mathbf{x} && (26)\\ \text{subject to} \quad & \mathbf{W}\mathbf{x} \le \mathbf{b} && (27)\\ & x_j \in N_0 \quad \forall j, && (28)\\ & \sum_{x_c \in C} x_c - \sum_{x_d \in D} x_d = 0 && (29)\\ & \mathbf{b} > \mathbf{0}, \mathbf{p} > \mathbf{0}, w_{ij} > 0 \quad \forall i, j, && (30)\\ & \mathbf{x}^T = (\mathbf{x}_C^T, \mathbf{x}_D^T), && (31) \end{aligned}$$

where x_C is the vector of item types of component class C and x_D is the vector of item types of component class D.

Note that, at least for GP, our simplifications are without restriction of generality and can be easily dropped as we discuss in Section 3.2.4. The system constraint is modeled by Equation (29) for 2-component systems; it must be appropriately adapted for the general case.

3.2.2 Test Data Set

The test data set chosen is the same as in the previous experiment. It is described in Section 3.1.2. For experiment 2a, we treat the first 50 types of items as members of component class C and the remaining 50 as members of component class D. In experiment 2b, the first 10 belong to component class C and the remaining 90 to component class D.

3.2.3 The Genetic Algorithm of Michalewicz (GA)

For the algorithm of Michalewicz, we use the version described in Section 3.1.3 with the parameters shown in Table 1 and a penalty function $P'(\mathbf{x})$:

$$P'(\mathbf{x}) = P(\mathbf{x}) + \rho' \cdot \left| \left(\sum_{x_c \in C} x_c - \sum_{x_d \in D} x_d \right) \right| \qquad (32)$$

$P(\mathbf{x})$ is given in Equation (15). The second term penalizes violations of the system constraint (29), where ρ' is a constant (set to 50 in the experiments).

Table 3: Test for H_0^k versus H_A^k in experiment 2a.

k	50	100	150	200	250	300
$\bar{x}_{GP,k}$	18225.08	19108.12	19564.27	19831.57	20017.84	20164.64
$\bar{x}_{GA,k}$	17212.99	17757.29	18054.32	18297.28	18481.15	18623.71
$s^2_{GP,k}$	219919.88	191182.30	170806.15	185494.70	206676.24	216147.43
$s^2_{GA,k}$	217428.74	206411.55	194283.01	200053.16	200316.03	206224.97
Z_k	34.22	47.90	55.88	55.25	53.86	53.02

Table 4: Test for H_0^k versus H_A^k in experiment 2b.

k	50	100	150
$\bar{x}_{GP,k}$	18051.99	18832.07	19225.37
$\bar{x}_{GA,k}$	9759.22	12617.35	14203.47
$s^2_{GP,k}$	168326.06	146895.43	171070.64
$s^2_{GA,k}$	10037992.16	7893595.85	5208914.12
Z_k	58.04	49.01	48.41
k	200	250	300
$\bar{x}_{GP,k}$	19480.24	19667.99	19819.74
$\bar{x}_{GA,k}$	15250.79	15962.36	16378.02
$s^2_{GP,k}$	208669.48	220428.95	226741.05
$s^2_{GA,k}$	3261738.05	2173218.10	1552639.26
Z_k	50.77	53.56	57.69

3.2.4 Genetic Programming with Linear Constraints (GP)

Grammar G_{2a} used in experiment 2a models the system constraint (29) in a natural way:

$$S \to K, \tag{33}$$
$$K \to A|AK|AKK, \tag{34}$$
$$A \to CD, \tag{35}$$
$$C \to a_1|a_2|\cdots|a_{50}, \tag{36}$$
$$D \to a_{51}|a_{52}|\cdots|a_{100}. \tag{37}$$

For experiment 2b, we replace the last two productions, (36) and (37), to obtain grammar G_{2b}:

$$C \to a_1|a_2|\cdots|a_{10}, \tag{38}$$
$$D \to a_{11}|a_{12}|\cdots|a_{100}. \tag{39}$$

For the linear constraint set, we simply add the (usual) definitions for $\mathbf{c}(s)$ and $\mathbf{r}(s)$ for the nonterminals C and D to Equations (21) and (22):

$$\mathbf{c}(C) = \mathbf{c}(D) = \mathbf{0} \tag{40}$$
$$\mathbf{r}(C) = \mathbf{r}(D) = \infty \tag{41}$$

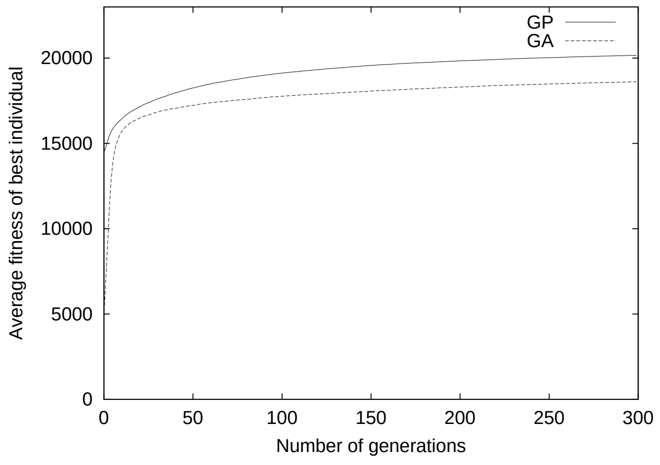

Figure 9: GA vs. GP for complementary goods (experiment 2a).

These production rules do constrain the search space, since only those derivation trees are valid that contain pairs of complementary items that form a system. Note that the production rules of the grammar can easily represent quite complex specifications, how systems should be composed, or what types of systems are possible.

3.2.5 Experiments 2a and 2b

The experimental setup for both experiments is the same as for experiment 1 in Section 3.1.5. However, experiments 1, 2a, and 2b differ in the way the system constraint (29) restricts the search: no restriction for experiment 1, a moderate restriction for experiment 2a (2500 complete systems), and an even tighter restriction for experiment 2b (900 complete systems). Tables 3 and 4 summarize the statistical test results. With $\alpha = 0.05$, all Z_k in both tables are larger than 1.65, and for all generations in both experiments, we have to reject the hypothesis that GA and GP have the same average performance. Figures 8, 9, and 10 tell the tale: average GP performance increases with respect to GA performance with the tightness of the system constraint.

Figure 11 graphically shows the effect of using a grammar in the experiments. For experiment 1, the grammar G_1 of the GP specified the same search space as Σ^* for the GA. All performance differences must, therefore, be attributed to differences in the structure of the σ-algebra of GP and GA whose characterization merits further investigation. However, in experiment 2a, grammar G_{2a} successfully models the system constraint (29) and thus reduces the search space of GP considerably. Experiment 2b is deliberately set up to demonstrate that grammar 2b achieves a further search space size reduction. The performance gains of GP in experiments 2a and 2b is, therefore, at least partially explained by the reduction of the search space by the grammars G_{2a} and G_{2b}.

4 Conclusion and Further Work

In this article, we have extended genetic programming over context-free languages to handle linear constraint systems in a straightforward and general manner. The algorithm was applied to two variants of the multidimensional knapsack problem, and

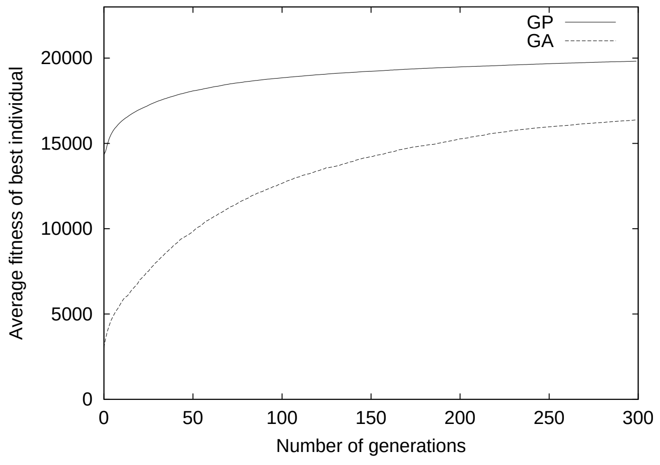

Figure 10: GA vs. GP for complementary goods (experiment 2b).

its performance was compared to Michalewicz's genetic algorithm with penalty functions. In the experiments reported in Section 3, genetic programming over context-free languages with linear constraints outperformed Michalewicz's genetic algorithm. Especially in early generations, a large discrepancy in the fitnesses of the best individuals was observed. We attribute this superior performance at least partially to a higher sampling rate in the feasible region, which increases the probability of finding better solutions.

This new grammar-based genetic programming variant is more flexible than its competitor, because all presented knapsack variants are easily handled within the framework of genetic programming over context-free languages with linear constraints. With the knapsack variant for systems (or complementary goods) we have discovered a new knapsack variant ideally suited for genetic programming over context-free languages with linear constraints, and it seems to be a promising approach for system assembly problems or modeling network effects between bundles of complementary goods (e.g., combinatorial auctions). In addition, we intend to extend this approach for multidimensional zero-one knapsack problems and compare it with Chu and Beasley's algorithm.

However, as usual, the current implementation still leaves several areas of improvement. In the algorithm for generating derivation trees, for example, the following options may improve performance:

- Keeping a table of a few small subtrees for each non-terminal symbol and, in case of a constraint violation, retry with one of these smaller subtrees or
- integrating a penalty function approach.

Variants of mutation and recombination algorithms that have not been studied are:

- If generation of a new subtree fails for a node, then choose another node and try.
- Various transformations of the probability of selecting an interior node for substitution by functions of tree-metrics (subtree size, depth, ...).

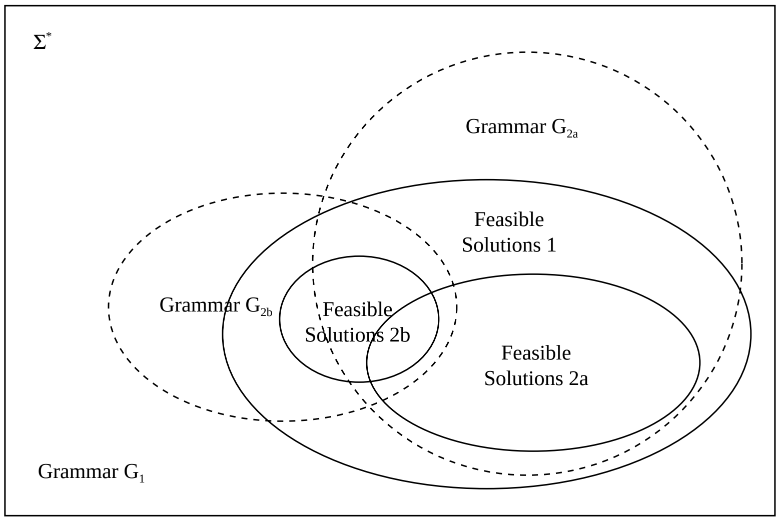

Figure 11: Feasible solution and search spaces in the experiments 1, 2a, and 2b.

- The mutation rate specifies the probability that a subtree is replaced. This would allow for more than one mutation operation per individual. Note, however, that a single mutation operation may affect a complete derivation tree and that mutation in a simple binary-coded genetic algorithm usually does not affect a complete chromosome.

For zero-one knapsack problems, set operations on the available terminal symbol sets should be implemented in order to replace the linear constraint system that enforces the zero-one condition on items. With this change, however, we leave the framework of context-free languages.

In all experiments, we used a grammar with the redundant production $K \rightarrow AKK$. The effect of this redundant rule on the performance of the algorithm is an average improvement of approximately one percent in the best solution found in all three experiments. However, even when using the grammar without the redundant production, genetic programming over context-free languages with linear constraints outperformed Michalewicz's genetic algorithm. For further research, we plan the analysis of redundant grammars and of the influence of normal forms on the performance of such algorithms, as well as the development of a version of an algorithm that coevolves redundant context-free grammars and the development of hybrid algorithms in the spirit of Chu and Beasley.

Acknowledgments

We acknowledge the support and help of three anonymous reviewers and of Ricardo Poli whose hints considerably improved the paper. We thank Anke Thede for correcting the revised manuscript and our production manager, Lisa Knebl, for the final edit of the paper.

References

Angeline, P. J. (1994). Genetic programming and emergent intelligence. In Kinnear, K. E., editor, *Advances in Genetic Programming*, Complex Adaptive Systems, pages 75–98, MIT Press, Cambridge, Massachusetts.

Antonisse, H. J. (1991). A grammar-based genetic algorithm. In Rawlins, G. J., editor, *Foundations of Genetic Algorithms*, pages 193–204, Morgan Kaufmann, San Mateo, California.

Auer, W. D. (1996). Ein Genetischer Algorithmus zur Lösung des zweidimensionalen Verschnittproblems. Diplomarbeit, Vienna University of Economics and Business Administration, Abteilung für Informationswirtschaft, Vienna, Austria.

Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm. In Grefenstette, J. J., editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 14–21, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Beasley, J. E. (1990). OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41:1069–1072.

Beasley, J. E. (1996). Obtaining test problems via internet. *Journal of Global Optimization*, 8:429–433.

Böhm, W. and Geyer-Schulz, A. (1997). Exact uniform initialization for genetic programming. In Belew, R. and Vose, M., editors, *Foundations of Genetic Algorithms*, Volume 4, pages 379–407, Morgan Kaufmann, San Francisco, California.

Bruhn, P. (2000). *Genetische Programmierung auf Basis beschränkter kontextfreier Sprachen zur Lösung kombinatorischer Optimierungsprobleme*. Dissertation, Vienna University of Economics and Business Administration, Abteilung für Informationswirtschaft, Vienna, Austria.

Chu, P. C. and Beasley, J. E. (1998). A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86.

Davis, L. (1991). A genetic algorithms tutorial. In Davis, L., editor, *Handbook of Genetic Algorithms*, pages 1–101, Van Nostrand Reinhold, New York, New York.

Dueck, G. (1993). New Optimization Heuristics: The Great Deluge Algorithm and the Record-to-Record Travel. *Journal of Computation Physics*, 104:86–92.

Gavish, B. and Pirkul, H. (1982). Allocation of databases and processors in a distributed computing system. In Akoka, J., editor, *Management of Distributed Data Processing*, pages 215–231, North Holland, Amsterdam, The Netherlands.

Geyer-Schulz, A. (1991). Fuzzy classifier systems. In Lowen, R. and Roubens, M., editors, *Computer, Management & Systems Science*, Volume 1, pages 86–89, IFSA, Brussels, Belgium.

Geyer-Schulz, A. (1993). Fuzzy classifier systems. In Lowen, R. and Roubens, M., editors, *Fuzzy Logic: State of the Art*, Series D: System Theory, Knowledge Engineering and Problem Solving, pages 345–354, Kluwer Academic Publishers, Dordrecht.

Geyer-Schulz, A. (1994). Fuzzy rule-based expert systems and genetic machine learning. Habilitationsschrift, Wirtschaftsuniversität Wien, Vienna, Austria. Appeared as Geyer-Schulz (1995).

Geyer-Schulz, A. (1995). *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, Volume 3 of *Studies in Fuzziness*. Physica-Verlag, Heidelberg, Germany.

Geyer-Schulz, A. (1996). *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, Volume 3 of *Studies in Fuzziness and Soft Computing*, Second revised edition. Physica-Verlag, Heidelberg, Germany.

Gilmore, P. C. and Gomory, R. E. (1966). The theory and computation of knapsack functions. *Operations Research*, 14:1045–1075.

Goldberg, D. E. (1987). Simple genetic algorithms and the minimal, deceptive problem. In Davis, L., editor, *Genetic Algorithms and Simulated Annealing*, Research Notes in Artificial Intelligence, pages 74–88, Pitman Publishing, London, UK.

Gottlieb, J. and Raidl, G. R. (2000). Characterizing locality in decoder-based eas for the multidimensional knapsack problem. In Fonlupt, C. et al., editors, *Proceedings of the Fourth Conference on Artificial Evolution*, pages 38–51, Volume 1829 of *LNCS*, Springer, Berlin, Germany.

Gruau, F. (1994). Genetic micro programming of neural networks. In Kinnear, K. E., editor, *Advances in Genetic Programming*, Complex Adaptive Systems, pages 495–518, MIT Press, Cambridge, Massachusetts.

Gruau, F. (1996). On using syntactic constraints with genetic programming. In Angeline, P. J. and Kinnear, K. E., editors, *Advances in Genetic Programming 2*, Complex Adaptive Systems, pages 377–394, MIT Press, Cambridge, Massachusetts.

Hillier, F. S. (1969). Efficient heuristic procedures for integer linear programming with an interior. *Operations Research*, 17:600–637.

Hoff, A., Løkketangen, A., and Mittet, I. (1996). Genetic algorithms for 0/1 multidimensional knapsack problems. Technical report, Molde College, Molde, Norway.

Hopcroft, J. and Ullman, J. (1979). *Introduction to automata theory, languages and computation*. Addison-Wesley, New York, New York.

Hörner, H. (1996a). A C++ class library for genetic programming: The Vienna University of Economics genetic programming kernel. Operating instruction, rel. 1.0, Vienna University of Economics and Business Administration, Abteilung für Informationswirtschaft, Vienna, Austria.

Hörner, H. (1996b). Ein Kern für genetisches Programmieren in C++. Diplomarbeit, Vienna University of Economics and Business Administration, Abteilung für Informationswirtschaft, Vienna, Austria.

Katz, M. L. and Shapiro, C. (1985). Network externalities, competition, and compatibility. *American Economic Review*, 75(3):424–440.

Keller, R. E. and Banzhaf, W. (1996). Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In Koza, J. R. et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 116–122, MIT Press, Cambridge, Massachusetts.

Khuri, S., Bäck, T., and Heitkötter, 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, New York.

Michalewicz, Z. (1994). *Genetic Algorithms + Data Structures = Evolution Programs*. Second edition. Springer, Berlin, Germany.

Montana, D. J. (1995). Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230.

Nguyen, T. and Huang, T. (1994). Evolvable 3D modeling for model-based object recognition systems. In Kinnear, K. E., editor, *Advances in Genetic Programming*, Complex Adaptive Systems, pages 459–476, MIT Press, Cambridge, Massachusetts.

O'Neill, M. (1999). Automatic programming with grammatical evolution. In O'Reilly, U.-M., editor, *Proceedings of the 1999 GECCO Conference*, Student Workshop.

O'Neill, M. and Ryan, C. (1999a). Automatic generation of caching algorithms. In Miettinen, K. et al., editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 127–134, John Wiley and Sons, Jyväskylä, Finland.

O'Neill, M. and Ryan, C. (1999b). Automatic generation of high level functions using evolutionary algorithms. In Ryan, C. and Buckley, J., editors, *Proceedings of the First International Workshop on Soft Computing Applied to Software Engineering*, pages 21–29, Limerick University Press, University of Limerick, Ireland.

O'Neill, M. and Ryan, C. (1999c). Genetic code degeneracy: Implications for grammatical evolution and beyond. In Floreano, D., Nicoud, J.-D., and Mondada, F., editors, *European Conference on Artificial Life*, pages 149–153, Springer, Berlin, Germany.

O'Neill, M. and Ryan, C. (1999d). Under the hood of grammatical evolution. In Banzhaf, W. et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 2, pages 1143–1148, Morgan Kaufmann, San Francisco, California.

O'Neill, M. and Ryan, C. (2001). Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358.

Powell, D. and Skolnick, M. M. (1993). Using genetic algorithms in engineering design optimization with non-linear constraints. In Forrest, S., editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 424–431, Morgan Kaufmann, San Mateo, California.

Raidl, G. R. (1998). An improved genetic algorithm for the multiconstrained 0-1 knapsack problem. In *Proceedings of the Fifth IEEE Conference on Evolutionary Computation, 1998 IEEE World Congress on Computational Intelligence*, pages 247–256, IEEE Press, Piscataway, New Jersey.

Raidl, G. R. (1999). Weight-codings in a genetic algorithm for the multiconstraint knapsack problem. In *Proceedings of the 1999 IEEE Conference on Evolutionary Computation*, pages 596–603, IEEE Press, Piscataway, New Jersey.

Richardson, J. T. et al. (1989). Some guidelines for genetic algorithms with penalty functions. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 191–197, Morgan Kaufmann, San Mateo, California.

Robert, C. P. (1999). *Monte Carlo Statistical Methods*. Springer, New York, New York.

Rothkopf, M., Pekec, A., and Harstad, R. (1998). Computationally manageable combinatorial auctions. *Management Science*, 44:1131–1147.

Rudolph, G. and Sprave, J. (1996). Significance of locality and selection pressure in the grand deluge evolutionary algorithm. In Voigt, H., Ebeling, W., and Rechenberg, I., editors, *Parallel Problem Solving from Nature – PPSN IV, Lecture Notes in Computer Science 1141*, pages 686–694, Springer, Berlin, Germany.

Ryan, C. and O'Neill, M. (1998). Grammatical evolution: A steady state approach. In Koza, J. R., editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, Stanford University Bookstore.

Ryan, C., Collins, J. J., and O'Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In Banzhaf, W. et al., editors, *Proceedings of the First European Workshop on Genetic Programming, Lecture Notes in Computer Science*, Volume 1391, pages 83–95, Springer-Verlag, Berlin, Germany.

Shih, W. (1979). A branch and bound method for the multiconstraint zero-one knapsack problem. *Journal of the Operational Research Society*, 30:369–378.

Starkweather, T. et al. (1991). A comparison of genetic sequencing operators. In Belew, R. K. and Booker, L. B., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 69–76, Morgan Kaufmann, San Mateo, California.

Thiel, J. and Voss, S. (1994). Some experiences on solving multiconstraint zero-one knapsack problems with genetic algorithms. *INFOR*, 32:226–242.

Whigham, P. A. (1995a). Grammatically-based genetic programming. In Rosca, J., editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Morgan Kaufmann, San Mateo, California.

Whigham, P. A. (1995b). Inductive bias and genetic programming. In Zalzala, A. M. S., editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, Volume 414, pages 461–466, IEE, London, UK.

Whigham, P. A. (1995c). A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation*, Volume 1, pages 178–181, IEEE Press, Piscataway, New Jersey.

Whigham, P. A. (1996a). *Grammatical Bias for Evolutionary Learning*. Ph.D. thesis, School of Computer Science, University College, University of New South Wales, ADFA.

Whigham, P. A. (1996b). Search bias, language bias, and genetic programming. In Koza, J. R. et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 230–237, MIT Press, Cambridge, Massachusetts.

Wong, M. L. and Leung, K. S. (1995). Inducing logic programs with genetic algorithms: The genetic logic programming system. *IEEE Expert*, 10(5):68–76.

Wong, M. L. and Leung, K. S. (1996). Learning recursive functions from noisy examples using generic genetic programming. In Koza, J. R. et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 238–246, MIT Press, Cambridge, Massachusetts.