Generating Lemmas for Tableau-based Proof Search Using Genetic Programming

Marc Fuchs
Fakultät für Informatik
TU München
80290 München, Germany
fuchsm@informatik.tu-muenchen.de

Dirk Fuchs
Fachbereich Informatik
Universität Kaiserslautern
67663 Kaiserslautern, Germany
dfuchs@informatik.uni-kl.de

Matthias Fuchs
Automated Reasoning Project
Australian National University
Canberra ACT 0200, Australia
fuchs@arp.anu.edu.au

Abstract

Top-down or analytical provers based on the connection tableau calculus are rather powerful, yet have notable shortcomings regarding redundancy control. A well-known and successful technique for alleviating these shortcomings is the use of lemmas. We propose to use genetic programming to evolve useful lemmas through an interleaved process of top-down goal decomposition and bottom-up lemma generation. Experimental studies show that our method compares very favorably with existing methods, improving on run time and on the number of solvable problems.

1 Introduction

Automated theorem proving (ATP) is an important area of artificial intelligence. There are two major paradigms for ATP that can be characterized as *bottom-up reasoning* and *top-down reasoning*. Usually, theorem provers centering on bottom-up reasoning—realized by *saturation-based* or *resolution-style* provers—infer consequences from the initial formulas (clauses) until an obvious inconsistency is derived, whereas top-down provers—often realized by *analytical* or *tableau-style* provers—attempt to recursively break down the proof goal until it is reduced to axioms. Both bottom-up and top-down provers apply given *inference rules* to create the search spaces that need to be traversed in search for a proof.

The search spaces in general are infinite which requires appropriate search-guiding heuristics to control the search. Both bottom-up and top-down reasoning have advantages and disadvantages that affect the efficiency of the search. Bottom-up reasoning can make use of strong simplification techniques to prune the search space (e.g., subsumption in resolution systems), but in its purest form lacks any kind of goal orientation. Top-down reasoning is per se goal oriented, but it suffers from redundant search effort since it is much more difficult to apply powerful simplification methods in the context of top-down reasoning (see [11]).

In a way, the strong points of top-down reasoning are the weak points of bottom-up reasoning and vice versa. Hence it makes sense to combine both methods in an appropriate way. For instance, the approach described in [16] employs a bottom-up prover, but restricts its inferences to certain relevant ones which are detected by top-down computations. In this paper we want to go another way and focus on top-down reasoning with the *connection tableau calculus* (CTC) [11]. Past research has demonstrated that introducing a bottom-up reasoning component into the CTC in the form of lemmas can be very profitable [14, 6, 5]. The difficulty of this approach is to provide a small number of lemmas that are reasonably likely to be useful. Current techniques for lemma generation as described in [14, 6] generate lemmas in a preprocessing phase and use the produced lemmas in the top-down proof run. Essentially, the lemma generation is a hill-climbing search for useful lemmas. Local decisions made during that process with the help of a *quality function* may prevent suitable lemmas from being generated.

Our solution to this problem is to use *genetic programming* (GP) [10] for the lemma generation in order to avoid local optima during the generation process. In using GP, we combine, similar to co-evolution, goal decomposition (i.e, breaking down the goal) with a saturation-based lemma generation. The fitness measure used is based on some notion of quality that utilizes similarity criteria. It guarantees that the lemma production influences the goal decomposition and vice versa. Thus, we introduce goal orientation into the bottom-up generation of lemmas. Since the top-down goal decomposition is also influenced by generated lemmas, the search is concentrated on "interesting" regions of the search space in a self-adaptive way. The performance of our approach is evaluated in the light of several examples taken from the TPTP library [17].

The paper is organized as follows. Section 2 outlines the basics of the CTC. Section 3 explains how lemmas can be utilized to alleviate its shortcomings. Our GP-based approach to create useful lemmas is described in Section 4 and is empirically evaluated in Section 5. Finally, a discussion in Section 6 concludes the paper.

2 Basics of the CTC

The CTC, like resolution, attempts to refute a set of *(input) clauses*. A clause C is a disjunction of *literals*: $C \equiv l_1 \vee \ldots \vee l_m$. Each literal is either a negative or a positive *atom*, i.e., $l_i \equiv \neg A$ or $l_i \equiv A$. $\sim l$ denotes A or $\neg A$ if $l \equiv \neg A$ or $l \equiv A$, respectively.

The CTC refutes a clause set $\mathcal{C}$ by constructing a *closed connected tableau* for $\mathcal{C}$. A *(clausal) tableau* is a tree where all non-root nodes are labelled with literals so that the following condition is satisfied: if the immediate successor nodes $v_1, \ldots, v_m$ of a node N are labelled with literals $l_1, \ldots, l_m$, respectively, then $l_1 \vee \ldots \vee l_m$ is (an instance of) a clause in $\mathcal{C}$. Henceforth, "node or leaf l" will be short hand for "node or leaf v labelled with l". A tableau is called *connected* if for each node v labelled with l (hence v cannot be the root) and successor nodes $l_1, \ldots, l_m$, there is at least one l_i so that $l_i \equiv \sim l$. Connected tableaux will also be called *connection tableaux* in the following. A tableau is *closed* if all branches of T are closed. A branch of T is closed if nodes l and l' occur on the branch such that $l \equiv \sim l'$. Otherwise, the branch is *open*. (A tableau with at least one open branch is open.)

The CTC employs the inference rules *start*, *extension*, and *reduction* to produce a closed connected tableau from a given set $\mathcal{C}$ of input clauses. The start rule can only be applied to the trivial tableau consisting of the root node only. It selects a *start clause* from $\mathcal{C}$ and attaches its literals (as nodes) to the root. It is not necessary to consider all clauses in $\mathcal{C}$ as possible start clauses. Commonly, only negative clauses (all literals are negative atoms) need be a start clause.

Extension attaches a clause $C \in \mathcal{C}$ to a leaf l of an open branch, so that the resulting tableau remains connected. That is, for at least one literal l' in C, there must be a *most general unifier (mgu)* σ such that $\sigma(l') \equiv \sigma(\sim l)$. σ is then applied to the whole tableau. Note that extension closes at least one branch. Reduction closes a branch of a tableau by finding a mgu σ of the leaf l and a literal l' elsewhere on the branch such that $\sigma(l) \equiv \sigma(\sim l')$. σ is again applied to the whole tableau (cp. [2]).

Finding a closed connected tableau is a search process that, for non-trivial problems, almost certainly involves inappropriate choices, in particular regarding extension and reduction steps. Hence, backtracking is required. However, since it is possible to construct infinite tableaux from a finite set $\mathcal{C}$, backtracking criteria other than "dead ends" are necessary.

To this end, *completeness bounds* [9, 15] are introduced which restrict the structure of tableaux that can be constructed. For example, the *depth bound* limits the depth, i.e., the maximal length, of a branch. Given a *resource* n for the depth bound, backtracking is employed as soon as an open branch exceeds this limit n. If all tableaux complying with the chosen completeness bound and resource have been enumerated without obtaining a closed tableau, the resource can be increased.

The literals at leaves of open branches are called *subgoals*. Let $l_1, \ldots, l_m$ be the leaf literals at the end of open branches of a tableau T. The clause $l_1 \vee \ldots \vee l_m$ is called the *subgoal clause* of T. The subgoal clause of T is a logical consequence of the input clauses (cp. [11]). The search scheme of the CTC can be viewed as systematically enumerating all subgoal clauses which can be derived from the original queries, i.e., the start clauses. More precisely, a tableau with an instance of a start clause below the unlabelled root node is called a *query tableau*. Such tableaux are enumerated by the CTC. The subgoal clauses of these tableaux are called *query clauses*. Clearly, the CTC is *goal oriented*, since each enumerated query clause has some connection with the original queries.

Goal orientation certainly is a strong point of the CTC. However, the CTC also has its weak points. The main shortcomings are long proof lengths and poor redundancy control. The proof length of a closed tableau T is equal to the number of inference applications to obtain T. Long proof lengths often are a consequence of repeatedly solving the same subgoal or instances thereof.

3 Lemmas in the CTC

To deal with the problems pointed out in the previous section, a bottom-up reasoning component in the form of lemmas can be employed.

Basically, lemmas can be seen as macro inference rules in the search space. A lemma originates from an open tableau and represents a sequence of inferences which transforms a subgoal into a (possibly empty) set of new subgoals. We use the following definition of a lemma which extends the notions given in [14, 1, 5].

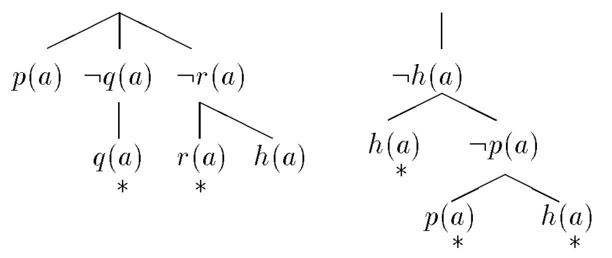
Definition 3.1 (lemma tableau, lemma clause) Let $\mathcal{C}$ be a clause set. Let T be a connection tableau for $\mathcal{C}$. Let $s_1 \vee \ldots \vee s_n$ be the subgoal clause of T. Let $\mathcal{H}$ be the set of the subgoals which are immediate successors of the root node. If $\mathcal{H} \neq \emptyset$ we call T a *lemma tableau*. If T is a lemma tableau, let s_i, $1 \leq i \leq n$, be the element of $\mathcal{H}$ which is left-most in T. We call $s_1 \vee \ldots \vee s_n$ the *lemma clause* of T. We write $s_i \leftarrow \sim s_1, \ldots, \sim s_{i-1}, \sim s_{i+1}, \ldots, \sim s_n$ and call s_i the *lemma head* and $\sim s_1, \ldots, \sim s_{i-1}, \sim s_{i+1}, \ldots, \sim s_n$ the *lemma tail*. In the following, a sentence $s_j \leftarrow \sim s_1, \ldots, \sim s_{j-1}, \sim s_{j+1}, \ldots, \sim s_n$ $(1 \leq j \leq n)$ is called a *contrapositive* of $s_1 \vee \ldots \vee s_n$. $\diamond$

Lemmas can be used like conventional input clauses during the inference process. An extension step with a lemma L (by unifying a subgoal with the lemma head) can be seen as implicitly attaching the lemma tableau for L below the subgoal.

The following example illustrates our notion of a lemma and the lemma use in the CTC.

Example 3.1 Let

$$\mathcal{C} = \{ \neg h(X), h(X) \vee \neg p(X), p(X) \vee \neg q(X) \vee \neg r(X), r(X) \vee h(X), q(a) \}$$



If the negative clauses are the start clauses, the left tableau is a lemma tableau (but no query tableau) representing the clause $p(a) \leftarrow \neg h(a)$ (which is equivalent to $p(a) \vee h(a)$). The right tableau, which is a query tableau, shows the application of the lemma in order to close the subgoal $\neg p(a)$ by extension with the lemma and performing reduction into the lemma tail. By using the lemma both proof length and proof depth can be reduced. $\diamond$

The example demonstrates that the use of lemmas as macro operators can reduce the proof depth and the proof length. Moreover, repeatedly solving certain subgoals can be avoided by using lemmas, which also reduces the proof length. Most of the completeness bounds which are of practical interest can profit from a proof length or depth reduction obtained with lemmas in the form of a resource reduction (see also [5]). This means that with lemmas the given input clauses can be refuted with a smaller resource value. However, the use of lemmas increases the branching rate of the search tree. More clauses may be applicable to specific subgoals and a lot of superfluous new inference possibilities may be introduced, thus enlarging the search space. One approach to alleviate these problems is the restriction of the lemma generation to *unit lemmas* which are lemmas with an empty tail. Such lemmas immediately close subgoals and do not introduce new subgoals.

Even when restricting the lemma use to units, there is still the problem to generate lemmas which are *relevant* for the given proof task, i.e., lemmas which can be used in a proof. A rather small set of lemmas which is sufficient to reduce the proof length can result in a dramatic speed-up of the proof search. It is in general not decidable whether a lemma is relevant for a given proof task. Therefore, heuristic criteria are applied. Lemmas are judged with respect to a quality function. Current lemma generation approaches use a preprocessing phase to generate lemmas by employing some saturation procedure controlled by a quality function (see e.g. [14]). The generated lemmas are then used in a final proof run. This proof run tries to refute the input clauses to which all (or some selected) generated lemmas are added.

The generation procedures have two serious difficulties. Firstly, the quality functions used to assess lemmas are rather vague. The quality of lemmas is judged without taking into account the current proof task. Quality criteria are, for instance, the number of symbols and the generality of a unit lemma. The criteria, however, ignore the proof goal. A consideration of query clauses occurring during the refutation attempt of the input clauses could improve the relevancy estimations. Secondly, the procedures basically perform some hill-climbing search for useful lemmas. Only the lemmas with high quality values persist. The generation of lemmas whose derivation requires the use of other lemmas, which are considered inappropriate, is impossible.

To overcome these difficulties, we introduce a novel GP-based approach for lemma generation. Our generation procedure works with connection tableaux as individuals and tries to generate lemma tableaux which are useful for a proof. In order to allow for better relevancy estimations of lemmas we also have query tableaux in the population. Our genetic operators are similar to conventional GP operators and extend the inference rules of the CTC. Our fitness measure—like the quality functions—is based on general purpose criteria like the number of symbols of the lemma or query represented by an individual. Additionally, we employ similarity criteria between open subgoals and lemmas in order to make the lemma generation more goal oriented. If a maximal number of generations is reached, lemmas are extracted from the final population.

4 The GP Approach

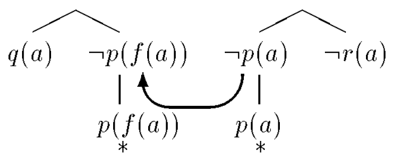
Our algorithm works on a population of fixed size. An individual corresponds to a connection tableau which represents a lemma or a query clause. We add type information to each tableau to clarify whether a tableau should be viewed as query or lemma tableau. This is necessary since a tableau may represent a query clause as well as a lemma clause. The initial population is created in the following way. The input clauses should be part of the initial population. This means that for each contrapositive of an input clause, a tableau of type 'lemma' is part of the population which represents the clause. Furthermore, for each start clause, a tableau is part of the population which is obtained by applying the start rule with the respective start clause. These tableaux are of type 'query'. In order to start with more difficult lemmas we may additionally use lemma clauses which are produced with conventional lemma generation algorithms (see also Section 5).

We employ three genetic operators, namely *reproduction*, a variant of *crossover*, and a kind of *mutation* operator in order to produce a successor generation. The operators are applied with fixed probabilities p_r, p_c, and p_m, respectively (see Section 5). The selected operator is applied to individuals of the old generation, which are selected probabilistically proportionate to their fitness, and produces new individuals which become a part of the successor generation.

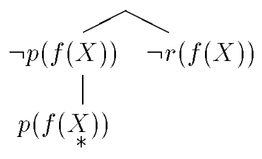
To ensure that each generation has available all lemma and query tableaux which represent input clauses, reproduction (which simply copies an individual) is applied to these tableaux as a first step when producing a successor generation. Deterministically copying these "designated" tableaux makes sense, because they are elementary "building blocks" for constructing connection tableaux, and should therefore be present in each generation. After that, the rest of the successor generation is obtained as described above.

Our crossover operator differs from standard crossover which exchanges two randomly chosen subtrees of two parent individuals. Since in many cases such an operation will not result in a connection tableau, crossover has to be constrained. A simple approach would be to allow crossover at nodes v_1 and v_2 (labelled with literals l_1 and l_2) of two individuals T_1 and T_2, respectively, only if $\mu = mgu(l_1, l_2)$ exists. Then, an exchange of the subtrees could take place and the resulting tableaux are instantiated with μ. This is certainly a reasonable method, but it neglects the fact that a re-use of a sub-deduction below v_1 in tableau T_1 may be possible below v_2 in T_2 although the above criterion is not applicable. We want to illustrate this with the following example.

Example 4.1 (Crossover) Let
$\mathcal{C} \supseteq \{q(a) \vee \neg p(f(a)), p(f(X)), p(a), \neg p(X) \vee \neg r(X)\}$.
The following figure shows two tableaux T_1 (left) and T_2 (right) for $\mathcal{C}$. The arrow (also called *link*) indicates that the sub-deduction below the literal $\neg p(f(a))$ at node v_1 in T_1 can be re-used below the goal $\neg p(a)$ at node v_2 in T_2. Although $\neg p(f(a))$ and $\neg p(a)$ are not unifiable, a reuse is possible, because we can *generalize* the sub-tableau below $\neg p(f(a))$. The sub-tableau can be generalized to a closed sub-tableau with root node $\neg p(f(X))$ (by applying the inferences in the sub-tableau to a most general goal $\neg p(X)$). Analogously, the query $\neg p(a)$ can be generalized by performing all inferences which have to be performed to create T_2, but omitting the inferences in the sub-deduction below $\neg p(a)$. This results in a more general query $\neg p(Y)$. Since $\neg p(Y)$ and $\neg p(f(X))$ are unifiable, the sub-deduction below node v_1 in T_1 can be used below node v_2 in T_2.



Crossover creates the following tableau by attaching the generalized sub-tableau below $\neg p(f(a))$ below the (generalized) goal. The resulting tableau is:



The sub-deduction below $\neg p(a)$ *cannot* be used below the goal $\neg p(f(a))$ in T_1 since no generalization can be performed and the literals are not unifiable. ◇

In summary, *asymmetric link relations* between tableaux can be set up that show which sub-deductions can replace others (see also [4]). Our crossover variant produces *one* new individual. In compliance with the link relation, in a *destination* tableau a (possibly empty) sub-tableau is replaced by a sub-tableau from a *source tableau*. Then, the modified destination tableau is instantiated in an appropriate manner (details can be found in [4]). In the above example, T_1 and T_2 serve as source and destination tableau, respectively. Crossover can be viewed as a *generalized extension step* which allows us to attach sub-tableaux rather than only clauses to (inner) nodes. Note that it is sensible to restrict the depth of tableaux which are produced by crossover (see Section 5).

Our mutation operator does not perform mutation in the usual sense. Instead, we use another kind of genetic operator with the aim to guarantee that each connection tableau can be created using only the genetic operators. In view of the fact that the start rule is represented by having in each generation all lemma and query tableaux corresponding to input clauses (see above), and that crossover is a generalized form of extension, we have to make sure that reduction steps can be performed. Therefore, we employ a genetic operator which allows for closing (inner) nodes of a tableau by reduction, and hence serves as a *generalized reduction operator*. Since reduction is an asexual operator, and a structural change of one individual takes place, we call this operation *mutation* or *reduction mutation*. A detailled description of the genetic operator can be found in [4].

The application of the genetic operators is controlled with an explicit fitness function which maps an individual to a real number. Smaller values are associated with fitter individuals. The abstract principles for measuring the fitness are as follows. Firstly, clauses should be rather small. If a query clause has a simple term structure and is quite general, it is more probable that the clause can be solved. Analogously, it is more probable that a general lemma is applicable in order to close open branches of a tableau. A "size part" Φ_{size} of the fitness is defined according to these criteria (see [4] for details).

Furthermore, we use a fitness component which is based on the similarity between a query and a lemma clause. If the literals of a query clause appear to be "almost solvable" (unifiable) with unit lemmas or with lemma heads (and the tail literals appear to be solvable by reduction), then the clause should be given a small fitness value. It is quite probable that such a goal decomposition is used in a proof. Conversely, lemmas which appear to be useful in order to close subgoals of goal decompositions may be helpful for refuting the input clauses. We employ a simple distance measure defined on literals l_1, l_2 as introduced in [4]. This measure considers certain syntactical properties of literals (*features*) and computes the Euclidean distance between the *feature-value vectors* of two literals.

We consider query tableaux to be useful if all subgoals are similar to unit lemmas. Due to efficiency reasons we do not consider non-unit lemmas, which would require a comparison of the open branches with the lemma tail literals. Lemma clauses are considered to be useful if the lemma head seems to be applicable to subgoals of queries and if the lemma tail is small, preferably empty. We refer the reader to [4] for details and an exact definition of the "similarity part" Φ_{sim}.

Finally, the fitness measure Φ is computed as a weighted sum of the size and the distance fitness value. Thus, we prefer tableaux which appear to be useful for the given proof task based on the currently available lemma and query information (small value of Φ_{sim}) as well as tableaux which may be useful based on general experiences (Φ_{size} small).

The evolutionary search stops if a query tableau can be extended to a closed tableau using the lemma tableaux, or a given maximal number of generations is reached. In the latter case all or some selected lemma clauses of the current population are used in a final proof run to refute the given input clauses.

5 Experimental Results

In the following we want to analyze the performance of our GP approach when assisting the prover SETHEO [12] which is based on the CTC. We examine whether the GP approach improves on conventional lemma generation techniques. Moreover, we investigate if our new theorem prover which uses a GP-based lemma preprocessor improves on the conventional prover without assistance by lemmas.

The test problems for examining these aspects are taken from the domains BOO, COL, GRP, and SET of the problem library TPTP [17]. We tackled only "hard problems". These are problems which cannot be solved with the conventional SETHEO system within 10 seconds. We used a SUN Ultra 2 and allowed for a run time of 15 minutes for each problem, which includes the time for the lemma generation as well as for the final top-down refutation run.

We employed three systems in our experiments. The first system is the conventional prover SETHEO as in [12]. The second system SETHEO/SAT additionally uses a conventional saturation-based lemma preprocessor which can be configured in various ways (described in [4]). In the experiments we used the best configuration found. Note that we consider a system which generates only unit lemmas. A final proof run which tackles the input clauses augmented by lemmas selected according to [5] is done with the same version of SETHEO as in our first variant.

The third system SETHEO/GP is based on our new GP-based lemma generator. At first, in a preprocessing phase, lemmas are generated with GP. The initialization is done as described in Section 4, i.e., the input clauses are used for the generation of lemma and query tableaux. We also add to the initial population a few lemma tableaux (a subset of the lemma set used by SETHEO/SAT) generated with the method given in [4]. Furthermore, we use additional query tableaux which are derived from the start clauses in a breadth-first search (see [4] for details). Reproduction of elements chosen randomly from the current partial population fills up the population until a given maximal size is reached. Thus, GP can operate from a well chosen starting point in the search space, and does not have to generate all lemmas from scratch. Note that with these initial lemmas worse results than with SETHEO/SAT would be obtained. After the initialization a GP run takes place. The specific configuration can be found in Table 1. Since the problems from the SET domain may contain up to 300 clauses, a large population size is needed there. After the generation of lemmas with GP, selection criteria are applied analogously to SETHEO/SAT (see also [5]) in order to select unit lemmas. Then, a run of SETHEO for refuting the input clauses plus the chosen lemmas takes place.

pop. size M_{max}	150, in SET theory 600
max. tableau depth d	7
reproduction prob. p_r	0.3
mutation prob. p_m	0/0.3 for Horn/non-Horn logic
crossover prob. p_c	$1 - p_r - p_m$
max. gen. no. G_{max}	15, in SET theory 7

Table 1: Configuration of the GP lemma preprocessor

The results of our experiments are listed in Table 2. We show the number of problems solved within 1, 5, and 15 minutes. The results of SETHEO and SETHEO/SAT are displayed in the first two columns. The results of SETHEO/SAT include the time for lemma generation and selection. The last two columns list the results of SETHEO/GP. "GP global" shows statistics regarding the total run-time of a prover consisting of the time for lemma generation with GP (including the time for creating the initial population), lemma selection, and the final proof run, whereas "GP top-down" gives statistics when omitting the time for lemma generation and selection. (The GP results are the best of 5 runs.) Since our actual implementation of GP is rather inefficient, in particular in SET long generation times are needed (about 80 seconds).

All lemma methods improve on the conventional SETHEO system in a rather stable way. Often, the use of these lemmas leads to a reduction of the resource needed to find a proof. Thus, considerably smaller search spaces have to be traversed. Nearly all of our successes result from reductions of the resource value needed and not from reordering effects of the search space within the smallest initial part of the search space which contains a proof.

When examining our results it becomes clear that, in spite of the fact that the lemma generation is more costly when using the GP-based approach, it improves on the other methods. In all of our domains we achieve either better or comparable results. Only in the COL

domain		SETHEO	SAT	GP global	GP top-down
BOO	≤ 1	5	11	11	11
	≤ 5	6	12	12	13
	≤ 15	11	13	14	14
COL	≤ 1	8	14	10	10
	≤ 5	10	29	27	27
	≤ 15	28	33	33	33
GRP	≤ 1	5	9	8	8
	≤ 5	9	11	15	15
	≤ 15	9	11	15	15
SET	≤ 1	18	40	0	43
	≤ 5	36	48	50	51
	≤ 15	41	50	54	54

Table 2: Experimental Results in the TPTP library

domain does GP lead to slightly longer run-times. 9 problems could be solved using GP-generated lemmas which are out of reach of the conventional lemma-generation techniques.

Preliminary tests have shown that GP itself can only solve trivial problems by closing query tableaux of the population with lemma tableaux. These problems can also be solved with a conventional prover within a few seconds. This is not surprising since we have to deal with tremendous search spaces. A solution of a hard problem with a small population size and a small number of generations as used in the experiments is unlikely. The use of GP as an "intelligent" lemma generator is nevertheless very profitable.

6 Discussion

Top-down theorem proving is a rather powerful methodology that can be improved significantly with lemma-generation techniques. In this paper we demonstrated that a GP-based lemma generation technique produces very promising results.

Compared to existing techniques, the population-based search of GP and its inherent random effects paired with a simultaneous evolution of top-down goal decomposition (query tableaux) and bottom-up lemma generation (lemma tableaux) are novel. In our opinion, these properties are very advantageous, because they make it possible to guide the search in a sensible way despite the vagueness and the heuristic nature of available knowledge.

Applications of GP to automated theorem proving (ATP) are quite scarce [8, 13, 7], and applications acknowledged by the ATP community are even rarer [3]. We believe that the method presented in this paper has the potential to make a significant contribution to ATP research.

References

[1] O.L. Astrachan and D.W. Loveland. The Use of Lemmas in the Model Elimination Procedure. *Journal of Automated Reasoning*, 19(1):117–141, 1997.

[2] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1996.

[3] M. Fuchs. Evolving combinators. In *Proc. CADE-14*, pages 416–430. LNAI 1249, 1997.

[4] M. Fuchs. An Evolutionary Approach for Combining Top-down and Bottom-up Proof Search. Technical report, TU München, 1998. See also `http://wwwjessen.informatik.tu-muenchen.de/ftp/Automated_Reasoning/Reports/AR-98-04.ps.gz`.

[5] M. Fuchs. Relevancy-Based Lemma Selection for Model Elimination using Lazy Tableaux Enumeration. In *Proceedings of ECAI-98*, pages 346–350. John Wiley & Sons, Ltd., 1998.

[6] M. Fuchs. Similarity-Based Lemma Generation for Model Elimination. In *Proceedings of CADE-15*, pages 33–37. Springer, LNAI 1421, 1998.

[7] M. Fuchs, D. Fuchs, and M. Fuchs. Solving problems of combinatory logic with genetic programming. In *Proc. GP-97*, pages 111–118. Morgan Kaufmann, 1997.

[8] T. Haynes, R. Gamble, L. Knight, and R. Wainwright. Entailment for specification refinement. In *Proc. GP-96*, pages 90–97. MIT Press, 1996.

[9] R. Korf. Macro-Operators: A Weak Method for Learning. *Artificial Intelligence*, 26:35–77, 1985.

[10] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[11] R. Letz, K. Mayr, and C. Goller. Controlled Integration of the Cut Rule into Connection Tableau Calculi. *Journal of Automated Reasoning*, 13:297–337, 1994.

[12] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. The Model Elimination Provers SETHEO and E-SETHEO. *Journal of Automated Reasoning*, 18(2), 1997.

[13] P. Nordin and W. Banzhaf. Genetic reasoning—evolving proofs with genetic search. In *Proc. GP-97*, pages 255–260. Morgan Kaufmann, 1997.

[14] J. Schumann. Delta - a bottom-up preprocessor for top-down theorem provers. System Abstract. In *Proceedings of CADE-12*. Springer, 1994.

[15] M.E. Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[16] M.E. Stickel. Upside-Down Meta-Interpretation of the Model Elimination Theorem-Proving Procedure for Deduction and Abduction. *Journal of Automated Reasoning*, 13:189–210, 1994.

[17] G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In *CADE-12*, pages 252–266, Nancy, 1994. LNAI 814.