Strategy Selection by Genetic Programming

Gernot Stenz & Andreas Wolf
Institut für Informatik der Technischen Universität München
D–80290 München
Germany
{stenzg,wolfa}@informatik.tu-muenchen.de

Abstract

Strategy parallelism is a powerful concept for applying parallelism to automated theorem proving. One of the most important problems to be solved in this approach is the proper distribution of the available resources among the different strategies. This task usually requires a lot of user expertise. When the resource distribution has to be done automatically, an adaptive algorithm must be used to optimize prover performance. We introduce a genetic algorithm that can be used for such an optimization and we show how such an algorithm can be integrated with other methods for automatic prover configuration. We give some experimental data to verify the validity of our approach and explain some of the future development possibilities.

Introduction

When applying parallelism to automated theorem proving there are several methods for splitting up the proof task. One of these methods, called *strategy parallelism*, distributes the workload by selecting a number of certain prover *strategies* and assigning resources (i.e. processors and time) to each of those. A strategy is a particular way of traversing the search space looking for a proof. This way we get a number of proof agents traversing the search space in parallel in different orders. The set of strategies together with their resource assignments is called a *schedule*. The task of devising an optimal schedule is highly non-trivial. Usually a lot of experimentation and user expertise is required to define which strategies are the most promising for a given problem and how the usually limited resources should be distributed among those strategies. However, an automated theorem prover usually is not applied to a single problem but to a larger set of problems where such manual prover tuning is not an option. Also, if the parallel prover is to be integrated as a sub-tool, for example into a system for software component retrieval (Fischer & Schumann 1997) or an interactive mathematical proof generation and presentation system (Dahn & Wolf 1996), the prover configuration must be done automatically without special knowledge of the problem domain. In order to achieve this goal one can employ adaptive techniques from the field of machine learning, like neural networks or genetic programming. We chose the latter to automatically generate semi-optimal schedules.

This paper is organized as follows. In the next section, we give an overview of the concept of strategy parallelism. Then we explain the problem of schedule optimization. This is followed by a brief introduction to genetic algorithms. After that we describe the combination of methods we employed for our solution. The subsequent section contains some data we obtained during our experiments, and finally, we give an assessment of our achievements and suggest future improvements.

Strategy Parallelism

For us, a *strategy* is one particular way of traversing the search space. Given our example of searching for connection tableau proofs, a strategy can be one specific way of using a completeness bound. We are now looking for a way of efficiently combining and applying different strategies in parallel.

Many ways of organizing parallel computing have already been proposed, developed and studied. However, many of these methods do not apply to automated theorem proving, since it is generally impossible to predict the size of each of the parallelized subproblems and it is therefore very hard to create an even workload distribution among the different agents. The different available approaches to parallelization in theorem proving can generally be divided into two categories: AND-parallelism and OR-parallelism.

With AND-parallelism, all subtasks have to be successfully completed to guarantee success for the entire problem, while in OR-parallelism it is sufficient if one subtask is successful. An example for AND-parallelism can be found in the *nagging* concept (Sturgill & Segre 1994): dependent subtasks will be sent by a master process to the naggers, which try to solve them and report on their success. The results will be integrated into the main proof attempt.

OR-parallel approaches include the *clause diffusion* concept of AQUARIUS (Bonacina & Hsiang 1993), where a resolution based prover with cooperating agents works on splitted databases. The cooperation of the distributed agents is necessary to guarantee completeness of the prover system. A combination of different strategies is used within the *Team-Work* concept (Denzinger 1995) of DISCOUNT (Denzinger*et al.* 1997). There a combination of several completion strategies is used for unit equality problems. These strategies periodically exchange intermediate results and work together evaluating these intermediate results and determining the further search strategies. *Partitioning* of the search space (Suttner & Schumann 1994) is done in PARTHEO (Schumann & Letz 1990). Partitioning *guarantees* that no part of the search space is explored more than once[1].

In (Cook & Varnell 1998) also an adaptive method for strategy selection for solving AI problems on finite but huge search spaces is used, which has some similarities to strategy parallelism.

Some of these approaches are very good in certain aspects. Partitioning, for example, can guarantee that no part of the search space is considered twice, therefore providing an optimal solution to the problem of generating "significantly" differing search strategies. The fundamental weakness of partitioning, however, is that all sub-provers must be reliable to guarantee the completeness of the prover system. Therefore, we have investigated a competition approach. Different strategies are applied to the same problem and the first successful strategy stops all others. Experimental experience shows that competition is a very good approach to parallelization in automated theorem proving (Schumann *et al.* 1998). However, not all strategies are equally promising or require equal effort. It is therefore advisable to divide the available resources in an adequate way.

The selection of more than one search strategy in combination with techniques to partition the available resources such as time and processors is called *strategy parallelism* (Wolf & Letz 1998; 1999). Different competitive agents traverse the same search space in a different order. Such a selection of strategies together with a resource allocation for the strategies is called a *schedule*[2].

The method of strategy parallelism implements a special kind of cooperation, *cooperation without communication* (Genesereth*et al.* 1986). Although the different agents do not communicate with each other after they have been started, they still cooperate since they have been chosen for their suitably different search space traversing behavior in the first place.

Strategy parallelism also avoids a fundamental disadvantage that is contained both in conventional AND- and OR-parallelization methods. In order to maintain completeness of the proof procedure, it is not necessary that *all agents* are reliable, a condition very difficult to ensure in a distributed environment. In contrast, strategy parallelism retains completeness as long as only *one agent* is reliable

The Multi-Processor Schedule Selection Problem

In strategy parallelism, we are faced with an optimization problem, the *strategy allocation problem (SAP)* which can be formulated as follows.

GIVEN a set $F = \{f_1, \ldots, f_n\}$ of problems, a set $S = \{s_1, \ldots, s_m\}$ of strategies, and two nonnegative integers t (time) and p (processors).

FIND a schedule $\{\langle s_1, t_1, p_1\rangle, \ldots, \langle s_m, t_m, p_m\rangle\}$ with $1 \leq p_i \leq p$ (strategy s_i will be scheduled for time t_i on processor p_i) such that

$$\left(\sum_{\{i:p_i=j\}} t_i \right) \leq t \text{ for all } j = 1, \ldots, p, \text{ and}$$

$$\left| \bigcup_{i=1}^{m} \{f \in F : s_i(f) \leq t_i\} \right| \text{ is maximal}^3.$$

We capture this problem by using a set of training examples from the given domain and optimizing the admissible strategies for this training set. The training phase, however, is extensive, as can be seen from the following consideration. Given a set of training problems, a set of usable strategies, a time limit, and a number of processors, we want to determine an optimal distribution of resources to each strategy, i.e., a combination of strategies which solves a maximal number of problems from the training set within the given resources. Unfortunately, even the single processor decision variant of this problem is strongly NP-complete[4]. Therefore, in practice, the determination of an optimal solution for the problem is problematic, and only suboptimal solutions can be expected. For manual prover configuration we determined a set of heuristically good schedules which are selected according to feature based problem analysis. The approach we used so far for automatic strategy selection is a combination of randomization elements with a gradient procedure of restricted dimension (Wolf 1998b). Even if the gradient procedure yields very good results, it should be mentioned

[1]There is also an AND-parallel (but less often used) variant of partitioning.

[2]Definition: Let $\{s_1, \ldots, s_n\}$ be a set of strategies. A *schedule* is a set S of ordered triples $\langle s_i, t_i, p_i\rangle$ $(1 \leq i \leq n)$ with the t_i (time) and p_i (processor ID) being non-negative natural numbers. The ordered pairs $\langle s_i, t_i, p_i\rangle$ are called *schedule components*.

[3]$s(f) = t$ means that the strategy s solves the problem f in time t.

[4]A problem is strongly NP-complete if it is NP-complete even if the numbers occurring in the input are written in unary notation. For details see (Garey & Johnson 1979). The SAP for one processor can be reduced to the *minimum cover problem* (Wolf & Letz 1999). The general case with an arbitrary number of processors additionally includes the *multiprocessor scheduling problem*, which is also known to be strongly NP-complete.

here that the complexity of the gradient algorithm is cubic in the size of its input. This complexity leads to long run-times on real problem and strategy sets. Another disadvantage of the gradient procedure is its bad adaptability to large sets of processors. The strategies are assigned to the processors at the beginning of the algorithm and can not be moved. So potentially badly performing strategy assignments can not be changed.

Genetic Algorithms

Genetic algorithms as a simple and efficient method of machine learning have been introduced in (Holland 1992; Koza 1992). A briefer introduction can be found in (Denzinger & Fuchs 1996).

Genetic algorithms maintain a set of suboptimal solutions for the given problem, called *individuals*. Each individual is described by a set of genes (a set of *attributes* and parameters). The set of individuals is called a *generation*. The fundamental approach of genetic algorithms is to evolve the set of individuals over a number of generations to obtain better solutions. This is done by transferring from one generation to the next the most successful or fittest individuals and replacing the least fitted individuals with new ones created from the preceding generation. For this, there are three different genetic operators.

The *reproduction* operator allows an individual to move on to the next generation. The *crossover* operator creates a new individual by mixing the genes of two parent individuals. This means that each attribute of the new individual is chosen from one of the attribute values of the parent individuals. Reproduction can be seen as a specialization of crossover with both parents being the same individual. Finally, *mutation* accounts for arbitrary changes in the gene set that may occasionally happen during reproduction. If an individual is selected for mutation, each attribute value is replaced with a certain probability by a randomly chosen new value. The operand individuals for each of these operators are chosen from the parent generation by applying a *fitness measure* to each individual. The initial generation can be created arbitrarily, usually by generating individual attributes at random.

The Genetic Gradient Approach

The *Genetic Gradient Approach* is the combination of the methods described in the previous two sections: A *genetic algorithm* is used to generate some preliminary results which are then further refined by some application of the *gradient method*.

In our approach the *individuals* are represented by prover schedules. The *attributes* of each schedule are pairs consisting of a processor and a time resource assignment for each strategy. The set of schedules constitutes a *generation*. The initial generation is created randomly, the subsequent generations are created by applying the genetic algorithm. As our *fitness measure*, we use the number of problems in the training set solved by an individual.

For our implementation, we chose the approach used in (Denzinger & Fuchs 1996) over the method of (Koza 1992) as it guarantees the survival of the most successful individuals of each generation: A predetermined portion of each generation is to die before each reproduction phase (controlled by a *kill-off-rate* parameter), the survivors make up the first part of the succeeding generation. This *truncate* or *elite selection* is followed by filling the remaining slots applying the crossover operator to the survivors. Mutations appear in the course of crossovers only, each newly generated individual is subject to mutations with a certain probability. This way each generation of individuals can be guaranteed to be at least as successful as the preceding one, which prevents the results from deteriorating over the generations. A setback of this method is that by reproducing the most successful individuals and allowing crossovers only with the reproduced individuals as parents such an algorithm tends to converge towards a small part of the attribute space. The element of mutation is used to counter this behavior to some degree but this second approach still does not cover the search space as good as the technique from (Koza 1992), with the benefit of a monotonously increasing fitness measure of the best individual over the number of generations.

The number of generations created is given by the user, but our results strongly suggest that, at least on a limited training set, after a certain number of generations no further improvements can be expected. We have empirical data for specifying that generation criterion. Termination criteria based on performance gains from one generation to the next do not lend themselves to our approach since it can be observed that the evaluated performance can stagnate over several generations to improve later on. For each generation the schedules are normalized. The normalization has to ensure that the accumulated time resources for each processor do not exceed the given limit. Furthermore, the full available amount of resources should be spent in the schedule. To follow these intentions, the normalization procedure for all processors p proportionally increases or decreases the time spent to each strategy assigned to p such that the consumption fits the resources.

The gradient method in combination with the genetic algorithm can be used to eliminate runaway strategies from schedules by transferring their time resources to more successful ones[5]. There are basically two ways to employ gradient optimization. First, it would be possible to use that method on each schedule as it is computed in each generation. But as the performance of schedules improves rather rapidly over the generations (as can be seen in the section contain-

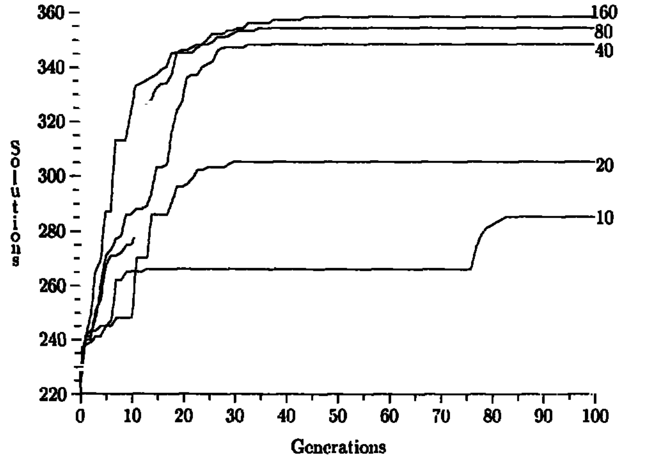
[5]For example, given a first-order problem, all resources assigned to strategies specialized on propositional problems would be wasted.

ing our experimental results) and gradient optimization is a very expensive procedure this does not seem recommendable. A second approach where the schedule evolution using the genetic algorithm is followed by a gradient optimization of the best resulting schedule combines good results with moderate cost.

Experimental Results

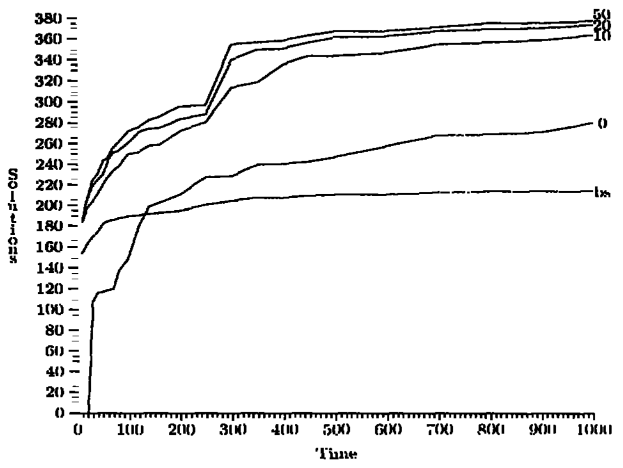
To evaluate our approach we used the 547 eligible TPTP problems (Sutcliffe *et al.* 1994) of the theorem prover competition at the 15th Conference on Automated Deduction 1998 to be our training set. The experimental data presented in this section was gathered in two distinct phases. Our participating prover p-SETHEO (Wolf 1998a) system employed 91 different strategies, these formed our strategy set. In the first phase, we extracted all 91 of p-SETHEO's strategies and ran each strategy on all problems using the standard sequential SETHEO (Moser*et al.* 1997). The successful results of all those runs were collected in a single list that became the database for our genetic gradient algorithm. 398 problems can be solved by at least one of the strategies in at most 300 seconds. Then, in the second phase we ran the genetic gradient algorithm on the collected data. The success of each of the schedules, as the individuals of our genetic algorithm, was evaluated by looking up the list entries for the problems and respective strategies and time resources.

In all experiments we used the gradient procedure and the genetic algorithm[6] described above. The attributes of the initial generation that are selected at random strongly influence the overall results of the experiment. The deficiencies of an unfit initial generation can not be wholly remedied by the subsequent optimizations. Therefore all experiments have been repeated at least ten times. The curves and tables depicted in this section represent the median results.

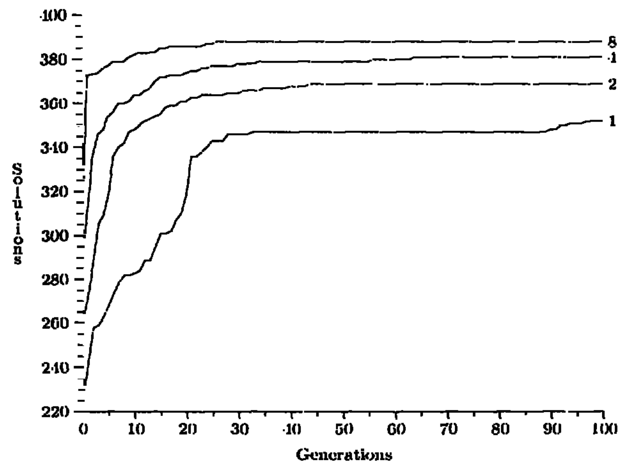


[6]The fixed parameters have been a kill-off rate of 0.6, a mutation rate of 0.1, and a mutation probability for each strategy of 0.2.

The first figure shows the number of problems solved after 0 to 100 generations for 10, 20, 40, 80, and 160 individuals (numbers at the curves) in 300 seconds on a single processor system.



The second figure displays the number of problems solved on a single processor system with 100 individuals after 0, 10, 20, and 50 generations (numbers at the curves) in the time interval from 0 to 1000 seconds. The behavior is compared with the best single strategy (denoted by *bs*). Note, that the strategy parallel system proved 378 problems within 1000 seconds, the best single strategy only 214, that is 57% of the strategy parallel version. These 214 problems have already been solved by the strategy parallel system after 25 (!) seconds.



The third figure illustrates the number of problems solved for 100 individuals and 100 generations with timeout values ranging from 0 to 1000 seconds on systems with 1, 2, 4, and 8 processors (numbers at the curves).

processors	1	2	4	8
gradient procedure (solutions)	355	361	374	382
genetic algorithm (solutions)	352	369	381	388

Finally, we compare the results of the gradient method and the genetic algorithm on 1, 2, 4, and 8 processors with 300 seconds each. We see the slightly better results of the gradient procedure on one processor. In all other cases, the genetic algorithm performs better.

Our experimental results show only a poor scalability for our actual prover system. That is due to the very limited number of training problems. Furthermore, many of the used strategies *overlap* one another (see (Wolf & Letz 1999)).

Assessment

In this paper we have shown the applicability of genetic programming methods to strategy parallel theorem proving. Especially in cases where prover tuning by the user is not an option and the system must automatically configure itself for a problem set, our approach improves on pre-selected single strategies. Compared to the old pure gradient method, the runtime needed decreases from hours to some minutes.

There are some points deserving our attention in future work. First of all, the number of strategies should be reduced to the point where only useful strategies remain in the strategy set; this means we only want to keep strategies that solve at least one problem that cannot be solved by any other strategy. Therefore, once we have obtained the prover performance data on the problem domain training set, a strategy evaluation and subsumption test might be useful.

Also, one of the greatest advantages of genetic algorithms, their generality, can turn out to be a major setback if genetic programming is applied unmodified to automated theorem proving. There, many problems fall into special domains (requiring a single special strategy and not a generic set of usually successful strategies) where the unbiased genetic approach will hardly yield good results. Additionally, a general approach will always waste resources on hopeless (at least in the average case) special strategies that might be used to better effect on standard strategies.

Undoubtedly, better results might be obtained if we combine the approaches of genetic programming and feature based problem analysis. There, instead of just one general schedule, using the genetic algorithm we compute a set of schedules for a certain set of problem categories distinguishable by syntactic and semantic problem features. In this case, the schedule for a special problem class, such as ground problems, will consist of the appropriate special strategies only, whereas the schedules for standard categories will still employ a mixture of several strategies. During an actual prover run, the problem is analyzed for its features and the selected schedule is executed.

Acknowledgments

This work is supported by the Deutsche Forschungsgemeinschaft within the Sonderforschungsbereich 342, subproject A5 (PARIS).

References

Bonacina, M. P., and Hsiang, J. 1993. Distributed Deduction by Clause Diffusion: The Aquarius Prover. *DISCO'93*, LNCS 722, 272–287.

Cook, D., and Varnell, C. 1998. Adaptive Parallel Iterative Deepening Search. *JAIR* 9:139–165.

Dahn, B. I., and Wolf, A. 1996. Natural Language Presentation and Combination of Automatically Generated Proofs. *FroCoS'96*, 175–192.

Denzinger, J., and Fuchs, M. 1996. Experiments in learning prototypical situations for variants of the pursuit game. *ICMAS'95*, 48–55.

Denzinger, J.; et al. 1997. Discount - a distributed and learning equational prover. *JAR* 18(2):189–198.

Denzinger, J. 1995. Knowledge-Based Distributed Search Using Teamwork. *ICMAS'96*, 81–88.

Fischer, B., and Schumann, J. 1997. SETHEO Goes Software Engineering: Application of ATP to Software Reuse. *CADE-14*, LNAI 1249, 65–68.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman.

Genesereth, M. R.; et al. 1986. Cooperation Without Communication. *AAAI-86*, 51–57.

Holland, J. H. 1992. *Adaptation in Natural and Artificial Systems.* MIT Press.

Koza, J. R. 1992. *Genetic Programming.* MIT Press.

Moser, M.; et al. 1997. SETHEO and E-SETHEO. The CADE-13 Systems. *JAR* 18(2):237–246.

Schumann, J.; et al. 1998. Parallel Theorem Provers Based on SETHEO. *Automated Deduction. A Basis for Applications. Volume II*, Kluwer, 261–290.

Schumann, J., and Letz, R. 1990. PARTHEO: A High-Performance Parallel Theorem Prover. *CADE-10*, LNAI 449, 40–56.

Sturgill, D., and Segre, A. 1994. A Novel Asynchronous Parallelism Scheme for First-Order Logic. *CADE-12*, LNAI 814, 484–498.

Sutcliffe, G.; et al. 1994. The TPTP Problem Library. *CADE-12*, LNAI 814, 252–266.

Suttner, C., and Schumann, J. 1994. Parallel Automated Theorem Proving. *PPAI'93*, 209–257.

Wolf, A., and Letz, R. 1998. Strategy Parallelism in Automated Theorem Proving. *FLAIRS-98*, 142–146.

Wolf, A., and Letz, R. 1999. Strategy Parallelism in Automated Theorem Proving. *IJPRAI* 13(2): to appear.

Wolf, A. 1998a. p-SETHEO: Strategy Parallelism in Automated Theorem Proving. *TABLEAUX'98*, LNAI 1397, 320–324.

Wolf, A. 1998b. Strategy Selection for Automated Theorem Proving. *AIMSA'98*, LNAI 1480, 452–465.