

# Simultaneous Search in Connection Tableau Calculi by Means of Disjunctive Constraints

Ortrun Ibens

Institut für Informatik, Technische Universität München, Germany  
ibens@informatik.tu-muenchen.de

## Abstract

Automated theorem proving with connection tableau calculi imposes search problems in tremendous search spaces. In this paper, we present a new approach to search space reduction in connection tableau calculi. In our approach structurally similar parts of the search space are compressed by means of disjunctive constraints. We describe the necessary changes of the calculus, and we develop elaborate techniques for an efficient constraint processing. Moreover, we present an experimental evaluation of our approach.

## Introduction

*Automated theorem proving* (ATP) is an important research area in artificial intelligence. The objective of an ATP system is to find out whether or not a *query* (or *goal*) is a logical consequence of a set of *axioms* (the query and the axioms have to be formally specified, for example in first-order clause logic). For this purpose, system-specific *inference rules* are applied systematically. A sequence of inference rule applications which shows that a given query is a logical consequence of a given set of axioms is called a *proof*.

The main strength of ATP systems is that they allow a purely *declarative* description of knowledge. However, the ability to handle declarative specifications introduces the aspect of *search* into the deduction process. The set of all objects which can be derived from an input problem by means of the inference rules forms the *search space*. In general, a tremendous search space has to be explored during the search for a proof. A large amount of the research performed in the field of ATP, therefore, focuses on search space reduction.

In this paper, we concentrate on search space reduction for *connection tableau calculi* (Letz, Mayr, & Goller 1994) which are successfully employed in ATP. For Horn input problems, search in connection tableau calculi is comparable with the interpretation of PROLOG programs. For non-Horn problems, there is an additional inference rule. Due to simplicity reasons, in this paper we will deal with the Horn case only. However, since the transfer to non-Horn input problems is straight forward (see (Ibens 1999)), the approach is applicable for ATP in full first-order logic. We are now going to describe how a certain kind of "redundancy

of the search" arises from structurally similar input clauses. This kind of redundancy can be tackled with our new technique.

In general, structurally similar clauses in the input to a connection tableau calculus lead to structurally similar computation trees. For example, in the input problem

$$S = \{ \neg p(X, Y) \vee \neg q(Y), \\ p(a_1, f(b_1)), \dots, p(a_n, f(b_n)), \\ q(g(d_1)), \dots, q(g(d_m)) \}$$

with  $n \geq 1$  and  $m \geq 1$ , the unit clauses in each of the sequences  $p(a_1, f(b_1)), \dots, p(a_n, f(b_n))$  and  $q(g(d_1)), \dots, q(g(d_m))$  only differ from each other in certain sub-terms. When given the initial computation tree

$$T_0 = \begin{array}{c} \wedge \\ \neg p(X, Y) \quad \neg q(Y) \end{array}$$

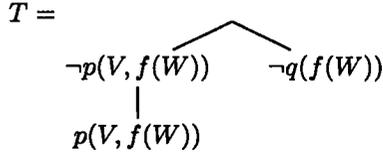
$n$  inferences are possible alternatively at the literal  $\neg p(X, Y)$ . These possible inferences involve the input clauses  $p(a_1, f(b_1)), \dots, p(a_n, f(b_n))$  and result in computation trees

$$T_i = \begin{array}{c} \wedge \\ \neg p(a_i, f(b_i)) \quad \neg q(f(a_i)) \\ | \\ p(a_i, f(b_i)) \end{array} \quad (\text{for } 1 \leq i \leq n).$$

The computation trees  $T_1, \dots, T_n$  only differ from each other in certain sub-terms occurring in the literals. Therefore, it seems redundant to construct each single computation tree  $T_i$  ( $1 \leq i \leq n$ ). A simultaneous processing of  $T_1, \dots, T_n$  seems more sensible. We now describe how this can be achieved.

The structurally similar unit clauses  $p(a_1, f(b_1)), \dots, p(a_n, f(b_n))$  can be compressed into the new clause  $p(V, f(W))$  with the additional condition  $\langle V, W \rangle \in \{ \langle a_1, b_1 \rangle, \dots, \langle a_n, b_n \rangle \}$ . The new clause expresses the common information of the structurally similar unit clauses, and the condition encodes their differences. The condition can also be viewed as defining the allowed instances of the clause  $p(V, f(W))$ . Thus, the clause set  $\{ p(a_1, f(b_1)), \dots, p(a_n, f(b_n)) \}$  is equivalent to the unit clause  $p(V, f(W))$  with its condition.

When given the above computation tree  $T_0$ , only one inference is possible at the literal  $\neg p(X, Y)$  when using the new clause instead of the compressed unit clauses. The resulting computation tree is



with the condition  $\langle V, W \rangle \in \{\langle a_1, b_1 \rangle, \dots, \langle a_n, b_n \rangle\}$ . The condition represents the allowed assignments to the variables occurring in the literals of  $T$ .  $T$  together with its condition can therefore be viewed as encoding the ordinary computation trees  $T_1, \dots, T_n$ .

Recapitulating, the computation trees  $T_1, \dots, T_n$  have been compressed into a single computation tree  $T$  with an additional condition. The compression has been performed by means of a compression of the structurally similar input clauses  $p(a_1, f(b_1)), \dots, p(a_n, f(b_n))$ . Since neither at the computation trees  $T_1, \dots, T_n$  nor at the computation tree  $T$  further inferences are possible, a pruning of the search space has been achieved. The number of tried inferences can be reduced even more if also the structurally similar input clauses  $q(g(d_1)), \dots, q(g(d_m))$  are compressed into the single clause  $q(g(Z))$  with the condition  $Z \in \{b_1, \dots, b_n\}$ .

In the example, there was no need to test the satisfiability of the conditions associated with computation trees. In general, however, a satisfiability test becomes necessary. In order not to simply move the search space from the computation trees into the satisfiability test, an intelligent handling of conditions is necessary. We first outline our new approach, then we present a detailed description and an evaluation of our approach. We conclude this paper with some comparisons with related approaches.

## Outlines of the new Approach

As explained above, input clauses are compressed before the proof run. The conditions arising from the compression of clauses are expressed by means of arbitrarily AND-OR-connected equations over first-order terms, called *constraints*.

If during the proof process a clause with a constraint is attached to a computation tree with a constraint, both constraints are instantiated, AND-connected, and the result is simplified. Since a constraint normal-form is not required, we use cheap simplification techniques which, on the one hand, can identify a lot of redundant or unsatisfiable sub-conditions but which, on the other hand, need not identify all redundant or unsatisfiable sub-conditions. The satisfiability test need not be performed after each inference. Since the satisfiability of a constraint follows from the existence of a solution, in the satisfiability test only one solution is computed.

This implies that *only one* of the simultaneously processed computation trees is computed explicitly.

The advantage of our approach over the conventional connection tableau calculus can be seen when regarding the example of the introduction. If a satisfiability test is applied to the condition  $\langle V, W \rangle \in \{\langle a_1, b_1 \rangle, \dots, \langle a_n, b_n \rangle\}$  associated with the computation tree  $T$ , then one solution (i.e., one allowed instantiation for  $\langle V, W \rangle$ ) can be found at low costs. After the failed inference step at the literal  $\neg q(f(W))$  all  $n$  represented computation trees are backtracked simultaneously.

## Compression of Clauses

We use constraints with the following syntax (henceforth, we always refer to this definition when using the denotation *constraint*):

### Definition 1 (Constraint)

1. If  $t$  and  $t'$  are terms, then  $t = t'$  is a *constraint*. We call  $t = t'$  an *elementary constraint*.
2. If  $c_1, \dots, c_k$ ,  $k \geq 0$ , are constraints, then  $c_1 \wedge \dots \wedge c_k$  and  $c_1 \vee \dots \vee c_k$  are *constraints*. For  $k \geq 2$ , we call  $c_1 \wedge \dots \wedge c_k$  a *conjunctive constraint* and  $c_1 \vee \dots \vee c_k$  a *disjunctive constraint*.  $\triangleleft$

If  $c$  is a conjunctive constraint  $c_1 \wedge \dots \wedge c_k$  or a disjunctive constraint  $c_1 \vee \dots \vee c_k$ , then  $c_1, \dots, c_k$  are *immediate sub-constraints of  $c$* . The notion of a *sub-constraint of a constraint* is inductively defined as follows:  $c$  itself is a *sub-constraint of  $c$* . If  $\hat{c}$  is an immediate sub-constraint of a constraint  $c$ , then  $\hat{c}$  is a *sub-constraint of  $c$* . If  $\hat{c}$  is a sub-constraint of  $c$ , then each immediate sub-constraint of  $\hat{c}$  is a *sub-constraint of  $c$* .

An elementary constraint  $t = t'$  is *valid* if and only if  $t$  equals  $t'$ . A conjunctive constraint  $c_1 \wedge \dots \wedge c_k$  is *valid* if and only if each  $c_i$  ( $1 \leq i \leq k$ ) is *valid*. A disjunctive constraint  $c_1 \vee \dots \vee c_k$  is *valid* if and only if there is a *valid*  $c_i$  ( $1 \leq i \leq k$ ). We denote the *instance* of a constraint  $c$  under a substitution  $\sigma$  by  $\sigma(c)$ . If  $c$  is a constraint and  $\sigma$  is a substitution such that  $\sigma(c)$  is *valid*, then  $\sigma$  is called a *solution* of  $c$ . A constraint is *satisfiable* if and only if a solution of it exists. Constraints  $c_1$  and  $c_2$  are *equivalent* if and only if each solution of  $c_1$  is a solution of  $c_2$  and each solution of  $c_2$  is a solution of  $c_1$ .

For a formal description of the compression of structurally similar clauses, we use the following definitions:

### Definition 2 (Generalization, Generalizer)

1. A clause  $C$  is a *generalization* of a set  $\{C_1, \dots, C_n\}$ ,  $n \geq 1$ , of clauses if and only if each clause  $C_i$  ( $1 \leq i \leq n$ ) is an instance of  $C$ . A set of clauses is called *generalizable* if and only if there is a generalization of it.
2. Let  $C$  be a generalization of a set  $S$  of generalizable clauses.  $C$  is a *most specific generalization* of  $S$  if and only if, for each generalization  $C'$  of  $S$ , there is a substitution  $\sigma$  such that  $\sigma(C') = C$ .

3. Let  $C, C_1, \dots, C_n$ ,  $n \geq 1$ , be clauses such that  $C$  is a generalization of the set  $\{C_1, \dots, C_n\}$ . Let  $\sigma_1, \dots, \sigma_n$  be substitutions such that, for each  $i$  with  $1 \leq i \leq n$ ,  $C_i = \sigma_i(C)$  and  $\sigma_i = \{X_{i,1} \leftarrow t_{i,1}, \dots, X_{i,n_i} \leftarrow t_{i,n_i}\}$  with  $n_i \geq 0$ . Then, we call the constraint

$$\bigvee_{i=1}^n (X_{i,1} = t_{i,1} \wedge \dots \wedge X_{i,n_i} = t_{i,n_i})$$

a *generalizer for  $\{C_1, \dots, C_n\}$  with respect to  $C$* . ◁

We regard clauses as structurally similar if and only if they are generalizable. During the compression of a set of structurally similar clauses a most specific generalization of the set and a generalizer for the set with respect to the most specific generalization are obtained. The most specific generalization yields the new clause resulting from the compression, and the generalizer yields the constraint of the new clause. For example,  $q(g(Y, Z))$  is a most specific generalization of the clause set  $\{q(g(a, a)), q(g(h(X), b_1)), \dots, q(g(h(X), b_m))\}$ , and

$$(Y = a \wedge Z = a) \vee \bigvee_{i=1}^m (Y = h(X) \wedge Z = b_i)$$

is a generalizer for this set with respect to  $q(g(Y, Z))$ .

According to the above definition, the generalizer for a set of structurally similar clauses with respect to a generalization of the set is always in *disjunctive normal-form*, i.e., it is a disjunctive constraint where each sub-constraint is a conjunction of elementary constraints. It can be transformed into an equivalent constraint with a minimal number of elementary sub-constraints by means of the commutative and distributive laws. The above generalizer, for example, can be transformed into the equivalent constraint

$$(Y = a \wedge Z = a) \vee (Y = h(X) \wedge \bigvee_{i=1}^m Z = b_i)$$

This transformation either reduces the number of elementary sub-constraints or leaves it unchanged. The number of elementary sub-constraints has an influence on the effort of the constraint simplification and the satisfiability test (see the next section).

### The New Calculi

During the construction of computation trees, constraints are handled as follows: If the start clause has a constraint, then its constraint results in a constraint of the initial computation tree. If during an inference rule application a clause with a constraint is attached to a computation tree with a constraint, the substitution which is applied to the clause and to the computation tree is also applied to both constraints. The constraint of the resulting computation tree results from a conjunction of the instances of both constraints. If the input computation tree does not have a constraint, then

the instantiated constraint of the input clause yields the constraint of the resulting computation tree. If the input clause does not have a constraint, then the instantiated constraint of the input computation tree yields the constraint of the resulting computation tree.

For conventional connection tableau calculi, a *closed* computation tree, i.e., a computation tree where each branch contains a pair of complementary literals, represents a proof. For connection tableau calculi with disjunctive constraints, however, a proof is represented by a closed computation tree with a satisfiable constraint. As shown in (Ibens 1999), connection tableau calculi with disjunctive constraints are sound and complete, i.e., a proof can be found in the new calculi if and only if the input clause set is unsatisfiable.

## Constraint Processing

We will now discuss techniques for the simplification and the satisfiability testing of constraints, and we will describe their use during the inference process.

**Constraint Simplification.** Valid or unsatisfiable sub-constraints can be identified and eliminated in certain cases.

According to the definition of being valid, valid sub-constraints of a given constraint can be identified by an analysis of elementary sub-constraints and a propagation of the results of the analysis. An analogous propagation technique allows the identification of certain unsatisfiable sub-constraints: We call a constraint *trivially unsatisfiable* if it is either an elementary constraint  $s = t$  where  $s$  and  $t$  are not unifiable, or a conjunctive constraint with a trivially unsatisfiable immediate sub-constraint, or a disjunctive constraint where all immediate sub-constraints are trivially unsatisfiable. Obviously, a trivially unsatisfiable constraint is unsatisfiable.

In the following cases valid or unsatisfiable sub-constraints may be eliminated from a given constraint. Let  $c$  be a constraint, let  $\bar{c}$  be a sub-constraint of  $c$ , and let  $\hat{c}$  be an immediate sub-constraint of  $\bar{c}$ . If  $\hat{c}$  is valid and  $\bar{c}$  is not valid, we can obtain a constraint  $c'$  which is equivalent to  $c$  by the elimination of  $\hat{c}$  from  $c$ . Analogously, if  $\hat{c}$  is trivially unsatisfiable and  $\bar{c}$  is not trivially unsatisfiable, we can obtain a constraint  $c'$  which is equivalent to  $c$  by the elimination of  $\hat{c}$  from  $c$ . Since the elimination of valid or trivially unsatisfiable sub-constraints of a constraint can significantly reduce the effort of testing its satisfiability, we remove valid or trivially unsatisfiable sub-constraints from the constraint associated with the current computation tree after each inference step.

**Satisfiability Test.** The satisfiability of a constraint follows from the existence of a solution of it. Therefore, only one solution of the given constraint is computed in the satisfiability test. The computation of a solution is based on the enumeration of solutions of elementary sub-constraints by means of backtracking —

until their composition is a solution of the whole constraint. A detailed discussion of different algorithms for the satisfiability test can be found in (Ibens 1999).

Obviously, an unsatisfiable constraint cannot become satisfiable by adding further sub-constraints. Therefore, inferences to a computation tree with an unsatisfiable constraint are not necessary; backtracking of inference steps can immediately be performed. However, due to the imposed effort, the satisfiability testing of constraints should be subject to accurate considerations. If after each inference the satisfiability of the constraint associated with the current computation tree is tested, then its possible unsatisfiability is immediately recognized. However, when performing the test after each inference, the accumulated effort of the satisfiability testing may be too high (in particular if most of the tested constraints are satisfiable). Another strategy may be to test the satisfiability of the constraint associated with a computation tree not until the computation tree is closed. In this case, however, a high number of useless inference steps may be applied to computation trees with unsatisfiable constraints.

In our approach, a satisfiability test is always applied after inferences involving unit clauses. These are situations where a certain sub-tree of the current computation tree has become closed. After inferences which involve a non-unit clause with a constraint a *weak satisfiability test* is applied only (see below).

**Weak Satisfiability Test.** If in an inference step a clause with a constraint is attached to a computation tree with a constraint, then according to the above-described inference rules the conjunction of both instantiated constraints yields the constraint of the resulting computation tree. Therefore, the constraint associated with a computation tree is in general a conjunctive constraint.

A weak satisfiability test of a conjunctive constraint can be performed if only one immediate sub-constraint is tested. If the tested immediate sub-constraint is unsatisfiable, then the whole constraint is unsatisfiable. Otherwise, neither the satisfiability nor the unsatisfiability of the whole constraint can be derived. In our approach, during the weak satisfiability test always that immediate sub-constraint is tested which has been attached in the recent inference step.

## Evaluation

We have integrated our approach into the automated connection tableau prover SETHEO (Letz *et al.* 1992; Ibens & Letz 1997; Moser *et al.* 1997). Thus we obtained the system C-SETHEO. In the following experiments, the performance of SETHEO and C-SETHEO is compared when run on different problems from version 2.1.0 of the benchmark library TPTP (Sutcliffe, Suttner, & Yemenis 1994) using a SUN Ultra 1 workstation (143 MHZ). The problems of the TPTP are formulated in first-order clause logic. When given an

Domain	Runtime	SETHEO	C-SETHEO
Field	≤ 10 sec	4	18
	≤ 60 sec	11	26
	≤ 150 sec	20	26
Planning	≤ 10 sec	0	1
	≤ 60 sec	1	5
	≤ 150 sec	3	6
Geometry	≤ 10 sec	0	4
	≤ 60 sec	2	9
	≤ 150 sec	5	10

Table 1: Numbers of solved non-trivial problems.

input problem, C-SETHEO first replaces each set of structurally similar input clauses with an equivalent pair consisting of a clause and a constraint. Then, it processes the result of this transformation. SETHEO tries to solve the original input problem directly.

We investigate problems from the field, planning and geometry domains. The TPTP library contains 281 field problems, 30 planning problems, and 165 geometry problems. We consider a problem as *trivial* if it can be solved by SETHEO as well as by C-SETHEO in less than 10 seconds. Thus, 55 of the field problems, 23 of the planning problems, and 50 of the geometry problems are considered trivial. Table 1 shows how many of the remaining problems can be solved within 10, 60, or 150 seconds. These time intervals are important since in applications like the software component retrieval (Dahn *et al.* 1997; Fischer, Schumann, & Snelting 1998; Baar, Fischer, & Fuchs 1999) answer times of 1 or 2 minutes are accepted in general.

For each of the tested domains and in each investigated time interval, C-SETHEO solves more problems than SETHEO. C-SETHEO even solves in 10 seconds nearly as many field or geometry problems as SETHEO in 150 seconds. That is, the integration of disjunctive constraints has achieved an important speed-up.

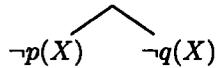
## Related Work

**Data-Base Unification.** The *data-base unification* (Bibel *et al.* 1998) has been developed as an improvement of the *connection method* (Bibel 1993) which is closely related to connection tableau calculi. In this approach, input clauses are compressed like in our approach. The arising conditions are expressed by specialized data-structures for the representation of substitutions, so-called *abstraction trees* (Ohlbach 1990).

If a clause with a condition is attached to a computation tree with a condition, the respective abstraction trees are merged by means of the operation *DB-join* (Bibel 1993). Since abstraction trees *explicitly* represent the allowed instantiations of the variables occurring in a condition, all redundant or unsatisfiable parts of the resulting condition have to be eliminated. This means that all satisfiable sub-conditions are de-

terminated. Thus, in almost each inference the data-base unification determines all ordinary computation trees which are represented by the current computation tree with its condition. In contrast to this, *only one* represented computation tree is determined in the satisfiability test of our approach. Therefore, in our approach considerably less computation trees are in general investigated during the search for a proof.

**Generalized Propagation.** *Generalized propagation* (Provost & Wallace 1992) integrates propagation techniques which have originally been developed for solving constraint satisfaction problems (Mackworth 1977) into the constraint logic programming scheme CLP(X) (Jaffar & Lassez 1987). The propagation of a literal is performed during the proof search, and it yields that information which have all answers to it in common. When given the computation tree



and the unit clauses  $p(a), p(g(b)), p(f(a)), q(f(a)), q(f(b))$ , for example, then  $\{X \leftarrow f(a)\}$  and  $\{X \leftarrow f(b)\}$  are the possible answers to the literal  $\neg q(X)$ . The common information of the computed answers is the fact that the variable  $X$  has to be instantiated with the term  $f(Y)$  where  $Y$  is a new variable. This information avoids that at the literal  $p(X)$  inferences with the input clauses  $p(a)$  or  $p(g(b))$  are tried.

An implementation of the concept resulted in the system PROPIA. In this system, propagation is only performed at certain propagation literals which have to be identified in the input clauses by the user. Since the effort and the benefits arising from the selection of a propagation literal can strongly differ for the literals of the input clauses, their determination requires the familiarity of the user with the behavior of the propagation process as well as with the formulation of the input problem. In contrast, our system works fully automatically. Another disadvantage of generalized propagation is that the propagation of a literal requires the computation of *all* answers to it in general. Therefore, it suffers from the repeated computation of sets of represented computation trees like the data-base unification.

## Conclusion

We have presented an approach to a simultaneous search in connection tableau calculi. This simultaneous search is achieved by the compression of structurally similar input clauses. The compression is performed by arbitrarily AND-OR-connected equations over first-order terms, called constraints. As a consequence of this compression a satisfiability test of the constraints associated with computation trees becomes necessary. In order not to move the search space which has been saved by the simultaneous search into the satisfiability test, elaborate strategies for the simplification and the

satisfiability testing of our constraints have been developed. An experimental evaluation shows that our approach increases the performance of the connection tableau prover SETHEO.

We have also discussed previous approaches aiming at a simultaneous search in connection-tableau-like calculi. As we have discussed, our new approach overcomes the deficiencies of the previous approaches and appears to be better suited in practice.

## References

- Baar, T.; Fischer, B.; and Fuchs, D. 1999. Integrating Deduction Techniques in a Software Reuse Application. *Journal of Universal Computer Science* 5(3):52-72.
- Bibel, W.; Brüning, S.; Otten, J.; Rath, T.; and Schaub, T. 1998. Compressions and extensions. In *Applied Logic Series 8*, 133-179. Kluwer Academic Publishers.
- Bibel, W. 1993. *Deduction: Automated Logic*. Academic Press, London.
- Dahn, B. I.; Gehne, J.; Honigmann, T.; and Wolf, A. 1997. Integration of Automated and Interactive Theorem Proving in ILF. In *Automated Deduction — CADE-14, LNAI 1249*, 57-60. Springer.
- Dincbas, M.; van Hentenryck, P.; Simonis, H.; Aggoun, A.; Graf, T.; and Berthier, F. 1988. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, 693-702.
- Fischer, B.; Schumann, J.; Snelting, G. 1998. Deduction-Based Software Component Retrieval. In *Applied Logic Series 10*, 265-292. Kluwer Academic Publishers.
- Ibens, O., and Letz, R. 1997. Subgoal Alternation in Model Elimination. In *TABLEAUX'97, LNAI 1227*, 201-215. Springer.
- Ibens, O. 1999. *Connection Tableau Calculi with Disjunctive Constraints*. Ph.D. Thesis, Institut für Informatik, TU München.
- Jaffar, J., and Lassez, J.-L. 1987. Constraint Logic Programming. In *Proc. of POPL'87*.
- Letz, R.; Schumann, J.; Bayerl, S.; and Bibel, W. 1992. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning* 8:183-212.
- Letz, R.; Mayr, K.; and Goller, C. 1994. Controlled Integration of the Cut Rule into Connection Tableau Calculi. *Journal of Automated Reasoning* 13:297-337.
- Mackworth, A. K. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8:99-118.
- Moser, M.; Ibens, O.; Letz, R.; Steinbach, J.; Goller, C.; Schumann, J.; and Mayr, K. 1997. SETHEO and E-SETHEO — The CADE-13 Systems. *Journal of Automated Reasoning* 18:237-246.
- Ohlbach, H. J. 1990. Abstraction Tree Indexing for Terms. In *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI-90)*, 479-484.
- Provost, T. L., and Wallace, M. 1992. Domain Independent Propagation. In *Proc. of FGCS-92*, 1004-1012.
- Sutcliffe, G.; Suttner, C. B.; and Yemenis., T. 1994. The TPTP problem library. In *Automated Deduction — CADE-12, LNAI 814*, 778-782. Springer.