

# Playing Games with Theorem Provers

-  
-  
Content Areas: Game Playing, Automated Reasoning, Theorem Proving  
-  
-

## Abstract

The disciplines of Computer Game Playing and Automated Theorem Proving have a lot in common and insights to share. From huge search spaces to heuristics and evaluation function, from learning to domain analysis to competition, both of these areas of Artificial Intelligence have much to gain from each other. This paper describes the use of a logic based language to describe games and a game playing system, named Monster, built from an automated theorem prover.

## 1 Introduction

Besides being fun, games play an important role in many aspects of computer science. From Logic to Machine Learning, Complexity Theory to Program Verification, games are used to aid in the analysis of complex problems.

Most research in Computer Game-Playing has been focused on a small number of specific games (most notably chess, checkers, backgammon, Othello and go). The result has been the development of programs capable of playing a single game very well, but unable to play any other games.

The recent presentation of a first-order logic based language for describing arbitrary games [Genesereth and Love 2004] leads naturally to the idea of using an automated theorem prover as the basis for a new game playing system.

This gives a basic introduction to computer game playing and a basic introduction to automated theorem proving. We then go on to point out some of the similarities between these two disciplines such as having to explore exponentially large search spaces, and the use of heuristics and evaluation functions to make these searches possible when given limited resources such as memory and time.

This paper also introduces Monster, a game playing program based on the automated theorem prover, JTP. We present some preliminary performance results and wrap up with some thoughts on how both ATP and computer game playing communities can benefit from each other.

## 2 Background

Before explaining why automated theorem proving might make a good starting place for a general game playing system, in this section we cover some basic information regarding game playing. We also outline existing formats for describing general games. This section is rounded out by outlining some basic information about automated theorem proving systems.

### 2.1 Game Playing Systems

Game playing systems all have one thing in common: search. A game can be represented by a game tree. In deterministic games, the nodes in such a tree correspond to game states, and the arcs correspond to moves. The initial state of the game is the root node; leaves of the tree correspond to terminal states.

By examining the game tree, all the way down to the terminal states, it would be possible to find the best strategy for playing any game. However, generating the entire game tree for complicated games, like chess, is completely infeasible. The game tree for chess would include some  $10^{120}$  positions which is thought to be greater than the number of atoms in the observable universe.

#### Evaluation

Since it is usually not possible to search the entire game tree, it is necessary to evaluate different position in order to choose the next move. This is done by using evaluation functions and search pruning strategies. The evaluation function returns a numerical estimate of the expected outcome of the game from a given position. The performance of a game playing program is dependent on the quality of the evaluation function. If it does not accurately reflect the actual chances of winning then the program will choose moves that lead to losing positions. The actual values of the evaluation function are not important so long as a better position has a higher value than a worse position.

Although some system have been successful using limited search depths [Epstein 2001; Tesauro 1995], for most systems a deeper search provides better play. Deep Blue uses special hardware to reach an average search depth of around 12 ply (or 6 moves). But to do so it narrows the search by pruning billions of possible positions [Campbell 1999] The evaluation function in conjunction with heuristics enable most game playing systems to prune unpromising paths from the game tree so the program does not waste time searching doubtful areas of the game tree.

#### Opening

One method of reducing the search space, and thereby increasing the playing strength, is the use of *opening books*. These are databases of opening moves which have been previously evaluated. The program can choose opening moves instantly because the best response has already been calculated ahead of time. Opening books have been quite successful for Chess [Campbell 1999] and Othello [Buro 2001].

*Endgame databases* are the same idea applied to the end of the game. They are successfully used in games where the number of choices is significantly reduced near the close of the game. Chinook, a world champion checkers program, has perfect

information for all checker positions involving 8 or fewer pieces [Lake et al. 1994]. That is over 400 billion positions.

Unfortunately, opening books and endgame databases require significant *a priori* information about, and analysis of, a particular game. The Chinook database has taken several years to compile. The general game playing problem presents a challenge in that the games aren't known beforehand. The best we could hope for is some time for analysis before the start of a new game. Through self play it will be possible to construct an opening book, although its size and utility may be limited.

## 2.2 Game Formats

A general game playing system needs a means of describing arbitrary games to the system. There have been a few methods proposed that do just that. Barney Pell created a game definition grammar, Metagame, and a game-playing program Metagamer [Pell 1993]. Pell's program plays a class of games called symmetric chess-like games. With the Metagame grammar it is possible to define games that have most of the rules of many games, including: chess, checkers, and tic-tac-toe.

Romien [Romein 2000] developed a language for expressing the rules of one and two person board games. The Multigame compiler generates a program that can play a game from a description of the rules. Orwant designed the EGGG language to codify game rules. The language is specifically designed to allow non-programmers to create games in with the Extensible Graphical Game Generator [Orwant 2000].

A class of games called Simple War Games was introduced in [Kaiser 2000]. The class includes both deterministic and non-deterministic games. While the search-space for Simple War Games are comparable to both chess and go, the grammar is not capable of fully describing games like chess or checker.

Recently, the Game Description Language (GDL) has been presented [Genesereth and Love 2004]. Designed specifically for future general game playing competition, the GDL is a subset of first order logic using syntax from the Knowledge Interchange Format (a.k.a. KIF language) [Genesereth 1998].

Unfortunately, each of these descriptive methods has limitations. Some are quite limiting. Metagame can only define games on a chess-like grid, Multigame can only define two person games and Simple War Games limit the types of piece movement.

## 2.3 Learning

The programs in [Pell 1993][Romein 2000][Kaiser 2000] are all able to learn to play by using the definition of the game, or game rules. Machine Learning in Game has a long history and a large body of reference material yet there is much to be explored. Furnkranz Machine Learning in Games bibliography contains over 400 entries. But relatively few deal with general game playing and fewer invoke automated theorem proving.

Several methods have had great success. Most successful programs, such as Deep Blue, Chinook and TD-gammon, use some kind of machine learning. Some, such as Deep Blue use machine learning techniques to fine tune their evaluation function. Others such as TD-gammon use techniques such as neural nets to actually construct the evaluation function. But it is an open question as to which learning method is best applied for each game situation. For example, Temporal Difference learning was used by Tesauro [1995] to produce a world champion backgammon program but it has been unsuccessful in many other games.

## 2.4 Automated Theorem Proving

Automatic Theorem Proving (ATP) deals with the development of computer programs to prove mathematical theorems. The language in which the theorems are written is a logic, often classical first order logic. Theorems are composed of axioms and hypotheses which will lead (or won't) to a conjecture. ATP systems produce proofs that describe how and why the conjecture is a logical consequent the axioms and hypotheses.

The proofs produced by ATP systems are intended to be readily understandable. For example, if the Towers of Hanoi puzzle were formulated as a theorem, the proof would describe the sequence of moves that need to be made in order to solve the puzzle.

ATP systems have been successfully used in many fields including software generation, software verification, security protocol verification and hardware verification. There are several ATP systems available. Some well known and successful first order logic systems are Otter, E, SPASS, Vampire and Waldmeister [Sutcliffe et al. 2000].

### Inference

A rule of inference is a pattern of reasoning consisting of one set of sentences, called premises, and a second set of sentences called conclusions. The following is a rule inference called Modes Ponens.

```
A -> B      if A is true then B is true
A           A is true
-----    therefore
B           B is true
```

ATP systems use a wide variety of inference strategies such unit resolution, linear resolution, set of support resolution, and term rewriting. The ATP system Vampire uses ordered binary resolution, superposition, and splitting [Voronkov 1994].

To prove a theorem the axioms of the theory to be proved are first put into a normal form called clausal form. An algorithm is then applied exhaustively to the resulting set of clauses in the search for a contradiction (the empty clause).

First order theorem provers such as Vampire use saturation algorithms [Riazanov and Voronokov 2003]. For each new clause generated by an inference the prover decides whether this clause should be kept or discarded. For non-trivial theorems a very large number of intermediate clauses need to be generated before the empty clause is found. This process can lead to a combinatorial explosion, so most systems perform inferences not on all kept clauses but only on a subset of them.

ATP systems essentially explore a tree of clauses, generated by their rules of inference, searching for the empty clause. But as is the case with game trees, generating a complete search tree can exhaust the available computational resources. Therefore, like game playing systems, ATP systems use heuristics (i.e. rules of thumb) for pruning inference steps and for guiding search through the space of inference steps.

In the E Equational Theorem Prover, search control heuristics define the order in which the prover considers newly generated clauses. A heuristic is defined by a set of clause evaluation functions and a selection scheme which defines how many clauses are selected according to each evaluation function [Schulz 2001].

The ATP system known as Gandalf is able to adapt its behavior. The system automatically selects search strategies which are likely to be useful for a given problem [Tammel 1997]. This is exactly the kind of behavior one would desire in a general game playing system.

## Competition

Comparisons between provers are based mostly on success rates during timed competition using standard collections of problems. The main collection is the Thousands of Problems for Theorem Provers (TPTP) library [Sutcliffe and Suttner 1998]. This is used as the basis for the annual CADE ATP System Competition (CASC), held at the Conference on Automated Deduction (CADE) [Sutcliffe 2001].

There are important differences in the types of problems. These differences have an impact on the techniques required to solving problems. For example a system that is able to solve satisfiability problems is typically not intended to solve unity equality CNF problems. Therefore CASC is organized into different divisions.

Each division uses problems that have certain logical, language and syntactic properties such as whether the problems are presented in first-order form FOF) or conjunctive normal form (CNF) and whether the problems are theorems, or a satisfiability (SAT) problems.

Whether or not equality is present in the problems is another important property. This has a major impact on both the logical features of problems and the algorithms employed to solve them.

There are many calculi that are complete for Horn clauses but incomplete for non-Horn clauses. Calculi that are not complete are unable to prove some true clauses. Whether or not the clauses of a CNF problem are all Horn clauses is another property used in CASC.

Each year a dozen or more ATP systems compete. Interestingly, no single system won more than two divisions. Some systems, such as the E Equational Theorem Prover are able to select strategies based on examining the problem given.

## 3 ATP and Games

Automated Theorem Proving and Computer Game Playing have a lot in common. And one would expect that competition would be an area where Computer Game Playing would shine.

As one might imagine there are also competitions in Computer Game Playing. The Computer Olympiad is an annual multi-game event in which all of the participants are computer programs. The purpose is to find the strongest programs at each of the games. It is presently under the auspices of the International Computer Games Association and the event hosts competitions in games such as Amazons, Chinese chess, go, and Hex.

Unfortunately there is no competition specifically for general game playing. Neither is there a nationally recognized collection of game definitions comparable to the TPTP library. This is clearly an area where the areas of machine learning and computer game playing can benefit from the ideas of the Automated Theorem Proving community.

### 3.1 Search and Evaluation

Exponential search space is the main problem facing both ATP and game-playing systems. Faced with limited resources, intelligently charting a course through this huge search space is the only means of answer a questioning in a reasonable amount of time. This is true whether the question is “Does the hypothesis follow from the premises?” or “Is there a winning move from this position?”

Both ATP systems and game playing systems use heuristics and evaluation functions to aid in searching exponentially huge search spaces.

With the recent appearance of a general game description language based on first-order logic (GDL), it seems only natural to

create a general game playing system using techniques from Automated Theorem Proving. In the next section we cover Monster, which is just such a system.

## 4 MONSTER

Monster is a game playing system based on the Java Theorem Prover (JTP) [Fikes et al. 2003]. JTP is a full first-order logic model elimination theorem prover developed by Gleb Frank. The system is an object-oriented, modular reasoning system based on a very simple and general reasoning architecture, which makes it easy to extend the system by adding new reasoning modules, or by rearranging or customizing existing ones. As the name implies, JTP is implement in Java.

In the JTP architecture, a reasoning system consists of modules called reasoners. Reasoners can be classified into two types, according to their functionality: backward-chaining and forward chaining. Backward-chaining reasoners process queries and return proofs for the answers they provide. Forward-chaining reasoners process assertions substantiated by proofs, and draw conclusions.

The principle data structure that reasoners provide in response to queries and assertion is a reasoning step, which is an object representing a proof for some sentence. Reasoning steps are often organized into a hierarchical proof tree.

Reasoners are the principle functional component of the architecture. A backward-chaining reasoner is an object that can accept goals and find proofs for the answers it returns. There are no formal limitations on what can be a goal for a backward-chaining reasoner. In the current JTP implementation, goals correspond to first-order logic sentences expressed in the conjunctive normal form.

The reasoner’s process function does most of the work. This method attempts to find proof for the goal. It returns an enumeration of reasoning steps that correspond to alternative proof for the goal.

The reasoning step is a unit of proof. Each reasoning step corresponds to an inference. Each reasoning step has a goal (the claim that is being proved). Compound reasoning steps have subgoals. Essentially, when a reasoning step has subgoals, it corresponds to an inference of the form:

```
Subgoal1, subgoal2, ... subgoalN
-----
Goal
```

A typical Modus ponens reasoning step can look like this:

```
(<= (foo ?x) (bar ?x)), (bar A)
-----
(foo A)
```

Compound reasoning steps can have sub-steps; each sub-step proves a particular sub-goal of this reasoning step. Therefore, together with all the requisite sub-steps, the reasoning step constitutes full proof for its goal. On the other hand, if a reasoning step is missing some of the sub-steps, such partial reasoning step is only a conditional proof, given the yet unproved subgoals.

The JTP architecture is representation language independent. However, the implemented JTP reasoning system uses the first-order logical representational language KIF. Theorems are stored in a single knowledge base as a set of symbolic sentence in conjunctive normal form (CNF).

## Playing Games

Monster first loads a game definition, described in GDL (which is a subset of KIF), and stores it into the knowledge base. The GDL version of Tic-Tac-Toe is three pages long. A portion of definition (in prefix KIF notation) is shown in figure 1. This segment determines which player has met the goal of three-in-a-row. The predicates for `column` and `diagonal` are similar to `row` and are not shown.

```
(<= (true (row ?n ?m) ?s)
    (true (cell ?n 1 ?m) ?s)
    (true (cell ?n 2 ?m) ?s)
    (true (cell ?n 3 ?m) ?s))

(<= (true (line ?m) ?s)
    (true (row ?n ?m) ?s))
(<= (true (line ?m) ?s)
    (true (column ?n ?m) ?s))
(<= (true (line ?m) ?s)
    (true (diagonal ?n ?m) ?s))

(<= (goal x ?s)
    (true (line x) ?s))

(<= (goal o ?s)
    (true (line o) ?s))
```

Figure 1 – Game Definition Language hypotheses for determining three-in-a-row in Tic-Tac-Toe.

After conversion to conjunctive normal form, the knowledge base will contain the clauses shown in Figure 2. Notice that the variables are renamed to be unique.

```
(or (true (row ?n_62 ?m_63) ?s_64)
    (not (true (cell ?n_62 1 ?m_63) ?s_64))
    (not (true (cell ?n_62 2 ?m_63) ?s_64))
    (not (true (cell ?n_62 3 ?m_63) ?s_64)))

(or (true (line ?m_65) ?s_66)
    (not (true (row ?n_67 ?m_65) ?s_66)))

(or (true (line ?m_68) ?s_69)
    (not (true (column ?n_70 ?m_68) ?s_69)))

(or (true (line ?m_71) ?s_72)
    (not (true (diagonal ?n_73 ?m_71) ?s_72)))

(or (goal x ?s_74)
    (not (true (line x) ?s_74)))

(or (goal o ?s_75)
    (not (true (line o) ?s_75)))
```

Figure 2 – CNF clauses stored in knowledge base.

At the start of each turn, Monster queries its database for legal moves, and selects one. The trick of course is selecting the best move. Normally, a game playing program would assign an evaluation function to each legal move, and explore the more promising moves by generating legal moves of the opponent. Alternating between its' own and opponent moves, the program would generate more moves until time runs out.

Guiding the search throughout would be the evaluation function. Unfortunately, the only distinguished predicates in GDL are *distinct*, *next*, *role*, *true*, *does*, *legal*, *goal* and *termination*. Notions about the structure of a particular game (like `cell`) or player movement (like `mark`) can not be relied on to be in the game definition.

For most board games, the structure of the game yields valuable information about strategy. For example, squares on a board can easily be seen as adjacent. By representing the board as a matrix or a graph much of this implicit information can be retained with minimal effort.

Additionally, in most game playing systems certain truths are held to be self evident. As with time or natural numbers, turn number two can be counted on to come after turn one. And like physics, a piece at rest stays at rest; pieces don't change position unless the players move them. Xs and Os aren't erased.

But nothing can be taken for granted in first order logic. Even the most basic facts must be explicitly listed. For example, the fact that the second turn comes after the first must be stated. Figure 3 shows how Game Description Language tells the fact that once a player puts their mark on the Tic-Tac-Toe board it stays there. It is also necessary to tell the system that a blank location stays blank if no players put a mark there.

```
;;;
;;; If a cell has a mark (not blank)
;;; on turn s then there will be a
;;; mark turn t (the next turn).
;;;
(<= (true (cell ?m ?n ?w) ?t)
    (next ?s ?t)
    (true (cell ?m ?n ?w) ?s))
    (distinct ?w b))

;;;
;;; A cell that is blank (b) on turn s
;;; will still be blank on the next
;;; turn (t) so long as a player
;;; marked a different location.
;;;
(<= (true (cell ?m ?n ?b) ?t)
    (next ?s ?t)
    (does x (mark ?m ?n) ?s)
    (true (cell ?m ?n b) ?s))
    (or (distinct ?m ?j)
        (distinct ?n ?k))

(<= (true (cell ?m ?n ?b) ?t)
    (next ?s ?t)
    (does o (mark ?m ?n) ?s)
    (true (cell ?m ?n b) ?s))
    (or (distinct ?m ?j)
        (distinct ?n ?k))
```

Figure 3 – Language Game Definition hypotheses for stating that a blank remains a blank and a mark (x or o) remains a mark from turn to turn.

Tic-Tac-Toe is a simple game with a relatively small state space. Even if we discount symmetry and other transformations there are only about 362,880 possible combinations of moves. However, due to the way games can be define in GDL, hundreds or even thousands of clauses must be created just to determine a legal

move. The farther along in the game, the more difficult it is to determine a legal move. On the first turn Monster only needs to generate forty three clauses to determine all the legal moves. But by the time both players have made two moves, Monster must generate over three thousand clauses to do the same task. Currently searching more than two ply deep is beyond Monsters abilities.

## 5 Testing

Pelletier [1986] published a list of seventy five problems in classical first-order logic for testing theorem provers. No such group of test problems for general game players exist. However the TPTP library contains over 7000 problems for automated theorem provers. This compendium contains Pelletier’s problems and a utility for converting problems into several languages used by ATP systems, among them KIF.

The library contains ninety problems in the domain of “Puzzles” and another thirty eight in the domain of “Planning (Blocks World)”. These, and possibly many others, would be ideal for testing Monster on single player games (puzzles).

The typical time given to ATP system to solve problems in CASC is between four and eight minutes. Monster is expected to operate in a much tighter time bound so many of TPTP will be completely out of range. Additionally Monster is intended to play games. Unless designed maliciously, the games themselves are not expected to be hard to play. Many problems from the TPTP library are not only hard to solve in a limited amount of time, but are in fact unsolvable. These would not be suitable for testing Monster.

In any case there is no utility for automatically converting TPTP problems into “games”. So this conversion is presently done manually. Unaltered TPTP problem can still be used to test Monster for soundness and correctness, but are unsuitable for testing Monster’s game playing abilities.

### 4.1 Experiments

We tested Monster on several problems from the TPTP v3.0.1 library as a benchmark. As these problems are not formulated as games they do not address Monster as a game playing system, but rather assess our implementation of the underlying JTP system. For comparisons we used Vampire version 7.0 and E version 0.62.

	difficulty			
	rating	Monster	Vampire	E
ANA006-1	.33	n/a	n/a	n/a
SYN036-2	.14	n/a	3	40
SYN303-1	.14	n/a	>1	>1
BOO027-1	.00	>1	>1	>1
PUZ001-3	.00	>1	>1	>1
PUZ054-1	.00	>1	>1	>1

Figure 4. Results of testing Vampire 7.0, E 0.82 and Monster on several problems from TPT v3.0.1. All times are in seconds. “n/a” indicates program was terminated after 200 seconds.

Some TPTP problems have difficulty ratings. These range from really easy (zero) to impossible difficult (1.0). The ratings are based on the performance of state of the art ATP systems at solving these problems. The ratings can change from year to year. Difficulty ratings for the chosen problems are shown in Figure 3. From the results in figure 4 it can be seen that Monster performs

fairly well on simple problems but is not very competitive on more complex problems.

## 6 Conclusion

This paper describes the use of a logic based language to describe games and an ATP system used to play them.

Automated Theorem Proving has a lot to offer the arena of game playing and visa versa. Techniques from Machine Learning in Games may offer insights into learning that can be applied to ATP systems to help them navigate through huge search spaces.

The collection of Thousands of Problems for Theorem Proving along with utilities to convert these problems into the languages used by various Automated Theorem Proving systems suggests that a similar collection would be useful in the area of general game playing.

The fact that different ATP systems excel in different categories of theorem proving suggests that general game playing would profit from the analysis of the basic structure of games and their classification.

The continuous improvement of ATP systems through annual competitions argues for similar competitions in general game playing.

The disciplines of Automated Theorem Proving and Computer Game Playing both have a lot to offer. The exchange of ideas and experience between these two communities will benefit both areas of Artificial Intelligence.

## References

- [Bundy 1999] Alan Bundy. A Survey of Automated Deduction. *Artificial Intelligence Today*. 153-174, 1999.
- [Buro 2001] M. Buro “Toward Opening Book Learning”, *ICCA Journal* 22(2) 1999, 98-102, reprinted in: *Games in AI Research*, H.J. van den Herik, H. Iida (ed.), ISBN: 90-621-6415-1, 2000, and in: *Machines That Learn to Play Games*, J. Furnkranz and M. Kubat (ed.), ISBN: 1-59033-021-8, 2001.
- [Campbell 1999] M. Campbell. *Knowledge Discovery in Deep Blue*, *Communications of the ACM* Vol 42 No 11 (November 1999) 65-67.
- [Epstein 2001] Susan Epstein. Learning to play expertly: a tutorial on Hoyle. In *Machines That Learn to Play Games*, J. Furnkranz and M. Kubat (ed.), ISBN: 1-59033-021-8, 2001.
- [Fikes et al. 2003] R. Fikes, G. Frank, J. Jenkins, *JTP: A System Architecture and Component Library for Hybrid Reasoning*. *Proceedings of the 7<sup>th</sup> World Multi-Conference on Systemics, Cybernetics and Informatics*. Orlando, Florida, USA. July 27-30, 2003.
- [Genesereth and Love 2004] Michael Genesereth, Nathaniel Love, *General Game Playing: Game Description Language Specification*, [unpublished] December 6, 2004, available at [ [http://games.stanford.edu/gdl\\_spec.pdf](http://games.stanford.edu/gdl_spec.pdf) ] as of 1/17/2005.
- [Genesereth, M. et. Al. 1998] Genesereth, M. et. Al. *Knowledge Interchange Format*,” draft proposed American National Standard (dpANS) NCITS.T2/98-004, <http://logic.sanford.edu/kif/dpans.html>

- [Lake et al 1994] Robert Lake, Jonathan Schaeffer and Paul Lu. Solving Large Retrograde Analysis Problems Using a Network of Workstations. Advances in Computer Chess VII, H.G. van den Herick, I.S. Herschberg and J.W.H.M. Uiterwijk (editors), University of Limburg, Maastricht, Netherlands pages 134-162, 1994.
- [Pelletier 1986] F.J. Pelletier (1986) "Seventy-Five Graduated Problems for Testing Automatic Theorem Provers" *Jour. Automated Reasoning* pp. 191-216.
- [Pelletier 1988] [Francis Jeffry Pelletier](#): Seventy-Five Problems for Testing Automatic Theorem Provers. 191-216 [BibTeX](#), Errata: JAR 4(2): 235-236 (1988)
- [Riazanov and Voronkov 2003] A. Riazanov, A. Voronkov, Limited Resource Strategy in Resolution Theorem Proving, *Journal of Symbolic Computation*, 36:1-2, 2003.
- [Schulz 2001] S. Schulz. System Abstract: E 0.61. Proceedings of First International Joint Conference on Automated Reasoning (IJCAR), Siena Italy, Jun 18-23, 2001, 370-375.
- [Sutcliffe and Suttner 1998] G. Sutcliffe, C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1, *Journal of Automated Reasoning* 21(2), pp.177-203. 1998
- [Sutcliffe et al 2000] G. Sutcliffe, M. Fuchs, C. Suttner. Progress in Automated Theorem Proving, 1997-1999. Technical Report TR-ARP-11-00, Australian National University, 2000.
- [Sutcliffe 2001] G. Sutcliffe (2001), The CADE-17 ATP System Competition, *Journal of Automated Reasoning* 27(3), pp.227-250.
- [Tammel 1997] T. Tammel. Gandalf. *Journal of Automated Reasoning*, Vol. 18, No. 2, 199-204, 1997.
- [Tesauro 1995] Gerald Tesauro. Temporal Difference Learning and TD-Gammon. In *Communications of the ACM*, March 1995, Vol 38, No. 3. 1995
- [Voronkov 1994] A. Voronkov, Implementing Bottom-up Procedures with code-trees: a case study of forward subsumption. Technical Report no. 88 October 3, 1994.