Implementation and Optimisation of a Tableau Algorithm for the Guarded Fragment

Jan Hladik

Theoretical Computer Science
TU Dresden

Abstract. In this paper, we present SAGA, the implementation of a tableau-based **S**atisfiability **A**lgorithm for the **Gua**rded Fragment ($\mathcal{GF}$). Satisfiability for $\mathcal{GF}$ with finite signature is EXPTIME-complete and therefore theoretically intractable, but existing tableau-based systems for EXPTIME-complete description and modal logics perform well for many realistic knowledge bases. We implemented and evaluated several optimisations used in description logic systems, and our results show that with an efficient combination, SAGA can compete with existing highly optimised systems for description logics and first order logic.

1 Preliminaries

The Guarded Fragment of first order predicate logic ($\mathcal{GF}$) [1] restricts the appearance of quantifiers to formulas of the kind

$$\begin{aligned} &\forall \boldsymbol{x}(G(\boldsymbol{x},\boldsymbol{y}) \rightarrow \varphi(\boldsymbol{x},\boldsymbol{y})) \\ \text{and}\quad &\exists \boldsymbol{x}(G(\boldsymbol{x},\boldsymbol{y}) \wedge \varphi(\boldsymbol{x},\boldsymbol{y}))\ , \end{aligned}$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are tuples of variables, and $G(\boldsymbol{x},\boldsymbol{y})$ is an atom, which is called the *guard* of the formula, whereas $\varphi(\boldsymbol{x},\boldsymbol{y})$ is called the *body*. This fragment has many desirable properties: satisfiability is decidable [1] in 2-EXPTIME, which is reduced to EXPTIME if the arity of the relations is bounded [12]. It also has the finite model property [1] and a tree model property for a special notion of a tree [12].

$\mathcal{GF}$ can be regarded as a generalisation of modal or description logics to n-ary relations (roles) [1, 10]. The more expressive such logics have a comparably high worst-case complexity, e.g. PDL [4] and $\mathcal{SHIQ}$ [23] are EXPTIME-complete [29, 22]. However, with optimised tableau algorithms like DLP [28], which decides PDL, and FaCT [19] or RACER [14], which decide $\mathcal{SHIQ}$, satisfiability becomes tractable for various realistic knowledge bases [19, 21, 15]. This suggests that a tableau algorithm for $\mathcal{GF}$-satisfiability might lead to an implementation that does not consume exponential time in practice.

$\mathcal{GF}$ is also decidable by resolution [9]. To the best of our knowledge, the efficiency of this approach for realistic formulas has not yet been analysed.

2 A Tableau Algorithm for $\mathcal{GF}$

In [17], a tableau algorithm is presented for the Clique Guarded Fragment ($\mathcal{CGF}$) [11], a generalisation of $\mathcal{GF}$. Its worst-case complexity is 2-NExpTime (NExpTime for signatures with bounded arity), and therefore higher than that of the automata algorithm in [12], but it allows for many of the optimisations known from description logics and therefore promises to lead to an efficient implementation.

Before describing this algorithm, we recall some definitions from [17]. Since our implementation Saga decides satisfiability of $\mathcal{GF}$ and not of $\mathcal{CGF}$, the definitions and the description of the tableau algorithm are restricted to $\mathcal{GF}$ formulas.

Definition 1 (NNF, closure). A formula $\varphi \in \mathcal{GF}$ is in *negation normal form (NNF)* if negation occurs only in front of atoms.

For a $\mathcal{GF}$-sentence φ in NNF, the *closure* $\mathrm{cl}(\varphi)$ is the set of all subformulas of φ. For a set C of constants, $\mathrm{cl}(\varphi, C)$ is the set containing all instantiations of $\mathrm{cl}(\varphi)$ with constants in C.

Table 1: Completion Rules for $\mathcal{GF}$

$\mathrm{R}\wedge$	if	$\varphi \wedge \chi \in \Delta(v)$
	then	$\Delta(v) := \Delta(v) \cup \{\varphi, \chi\}$
	unless	$\{\varphi, \chi\} \subseteq \Delta(v)$
$\mathrm{R}\vee$	if	$\varphi \vee \chi \in \Delta(v)$
	then	$\Delta(v) := \Delta(v) \cup \{\psi\}$ for a ψ chosen non-deterministically from $\{\varphi, \chi\}$
	unless	$\{\varphi, \chi\} \cap \Delta(v) \neq \emptyset$
$\mathrm{R}\dot{=}$	if	$a \doteq b \in \Delta(v)$
	then	for all nodes w that contain a:
		$C(w) := (C(w) \setminus \{a\}) \cup \{b\}$ and $\Delta(w) := \Delta(w)[a \mapsto b]$
	unless	$a = b$
$\mathrm{R}\forall$	if	$\forall \boldsymbol{x}(G(\boldsymbol{x}, \boldsymbol{c}) \rightarrow \varphi(\boldsymbol{x}, \boldsymbol{c})) \in \Delta(v)$,
		and there exists a $\boldsymbol{b} \subseteq C(v)$ such that $G(\boldsymbol{b}, \boldsymbol{c}) \in \Delta(v)$
	then	$\Delta(v) := \Delta(v) \cup \{\varphi(\boldsymbol{b}, \boldsymbol{c})\}$
	unless	$\varphi(\boldsymbol{b}, \boldsymbol{c}) \in \Delta(v)$
$\mathrm{R}\exists$	if	$\exists \boldsymbol{x}(G(\boldsymbol{x}, \boldsymbol{c}) \wedge \varphi(\boldsymbol{x}, \boldsymbol{c})) \in \Delta(v)$
	then	let $\boldsymbol{b}$ be a sequence of new constants with the same length as $\boldsymbol{x}$
		create a son w of v with $C(w) := \boldsymbol{b} \cup \boldsymbol{c}$, $\Delta(w) := \{G(\boldsymbol{b}, \boldsymbol{c}), \varphi(\boldsymbol{b}, \boldsymbol{c})\}$
		$N(w) := 1 + \max\{N(v) : v \in V \setminus \{w\}\}$
	unless	there is a $\boldsymbol{b}, \boldsymbol{c} \subseteq \Delta(v)$ with $\{G(\boldsymbol{b}, \boldsymbol{c}), \varphi(\boldsymbol{b}, \boldsymbol{c})\} \subseteq \Delta(v)$,
		there is a son w of v with $\{G(\boldsymbol{b}, \boldsymbol{c}), \varphi(\boldsymbol{b}, \boldsymbol{c})\} \subseteq \Delta(w)$ for some $\boldsymbol{b}, \boldsymbol{c} \subseteq \Delta(w)$,
		or v is blocked
$\mathrm{R}\updownarrow$	if	$\varphi(\boldsymbol{c}) \in \Delta(v)$ is an atomic or universally quantified formula,
		and w is a neighbour of v with $\boldsymbol{c} \subseteq C(w)$
	then	$\Delta(w) := \Delta(w) \cup \{\varphi(\boldsymbol{c})\}$
	unless	$\varphi(\boldsymbol{c}) \in \Delta(w)$

The $\mathcal{GF}$ tableau algorithm operates on a *completion tree*, a vertex labeled tree in which every node stands for a set of constants which appear together in a guard atom.

Definition 2 (Completion Tree, Blocking, Tableau). Let $\varphi \in \mathcal{GF}$ be a sentence in NNF. A *completion tree* $T = (V, E, C, \Delta, N)$ for φ is a vertex labeled tree (V, E) with the function C labeling each node $v \in V$ with a set of constants, Δ labeling each $v \in V$ with a subset of $\mathrm{cl}(\varphi, C(v))$, and N mapping each node to a distinct natural number such that if v is an ancestor of w, then $N(v) < N(w)$.

A node $v \in V$ is called *directly blocked* by a node $u \in V$ if u is not blocked, $N(u) < N(v)$, and there is an injective mapping π from $C(v)$ to $C(u)$ such that for all constants $c \in C(v) \cap C(u)$, $\pi(c) = c$ and for the extension of π to formulas, $\pi(\Delta(v)) = \Delta(u)|_{\pi(C(v))}$, where $\Delta(u)|_{\pi(C(v))}$ denotes the set of all formulas in $\Delta(u)$ containing only constants in $\pi(C(v))$. A node is called *blocked* if it is directly blocked or if its father is blocked.

A completion tree T *contains a clash* if there is a node $v \in V$ such that $\neg(c \doteq c) \in \Delta(v)$ for a constant $c \in C(v)$ or there is an atomic formula φ such that $\{\varphi, \neg\varphi\} \subseteq \Delta(v)$. Otherwise, T is called *clash-free*. A completion tree T is called *complete* if none of the *completion rules* given in Table 1 can be applied to T. A *tableau* is a complete and clash-free completion tree.

For a formula set Φ and constant symbols a, b, we use the notation $\Phi[a \mapsto b]$ to denote the set of formulas in Φ where all occurrences of a are replaced by b.

The blocking condition is *dynamic*, i.e. blockings are not established forever, but they can be canceled later if one of the nodes involved changes (and reappear if the other one changes accordingly).

To decide the satisfiability of a formula φ, a root node n_0 is created with $N(n_0) := 0$, $C(n_0) := \{c\}$ for a random constant c (to prevent empty structures), and $\Delta(n_0) := \{\varphi\}$. Then the completion rules in Table 1 are applied until a tableau is found or a clash occurs. If the rules can be applied in such a way that a tableau is found, "φ `is satisfiable`" is output, otherwise "φ `is unsatisfiable`".

Since a satisfiable $\mathcal{GF}$-formula need not have a *finite tree* model, the blocking condition is necessary to ensure termination. It prevents new nodes from being created if there is already another node containing "similar" formulas. If a tableau is found, it can be transformed into a model by *unraveling* the blockings, i.e. replacing a blocked node v with a copy of the node u blocking v. Note that the blocking condition is not equivalent to *subset blocking*, where a node v is blocked by a node u if the label of v is a subset of the label of u: for $\mathcal{GF}$, the image of π need only be a subset of $C(u)$, but restricted to these constants, the labels of u and v have to be equal (modulo π). The proof of correctness, completeness and termination of the algorithm can be found in [17].

3 Implementation

The algorithm described in Section 2 leaves many possibilities for an implementation. To obtain an efficient program, the following issues have to be considered:

Non-Determinism. There are two kinds of non-determinism involved in the $\mathcal{GF}$-algorithm: the decision which rule to apply first if several ones are applicable is *don't-care* non-deterministic, i.e. every choice will lead to a correct behaviour of the algorithm, but its efficiency will depend on a good heuristic. The two heuristics implemented in SAGA are described in Section 5.

The decision which disjunct of a disjunction to add to the corresponding node is *don't-know* non-deterministic, i.e. only certain choices will lead to the discovery of a tableau. Therefore, a *branching* and *backtracking* technique is necessary to undo the changes made by the last decision after a clash has occurred. Efficiency will again depend on a good strategy which disjunct to try first.

The data structure *branching point* described in Section 3 is used to enable branching and backtracking: before adding the first disjunct φ to the corresponding node, a new branching point is created that subsequently stores backups of all nodes that are changed as a consequence of adding φ. Three different heuristics for choosing the first disjunct are implemented (cf. Section 5).

Data Structures. The data structure for a node n contains the labels $C(n)$, $\Delta(n)$, and $N(n)$ described in Section 2. Additionally, blockings that were already detected are recorded in the blocking node as well as in the blocked one such that unnecessary testing is avoided.

A branching point b is created by R$\vee$ after choosing a disjunct ψ from a disjunction $\varphi \vee \chi$. It contains a unique *branching identifier (BID)* $I(b)$, a list $C(b)$ of nodes that were created, and a list $M(b)$ of backups of nodes that were modified as a consequence of adding φ, and the other disjunct $O(b) = \chi$ that has to be added to the tree if φ causes a clash.

For a node n, every formula $\varphi \in \Delta(n)$ is labeled with the *dependency set* $D(\varphi, n)$ of branching points it depends on. This enables us to find the right branching point for the backup of a node n if a rule application for φ modifies n, and to use *backjumping* (cf. Section 4).

Functions. Figure 1 shows the function hierarchy: an arrow from `f1` to `f2` means that `f1` invokes `f2`. The main function `construct-tableau(`φ`)` receives a $\mathcal{GF}$ formula as input. It creates a new node n_0 and adds φ to $\Delta(n_0)$. Subsequently, it iteratively calls `choose-next-formula()`, which uses one of the heuristics described in Section 5 to determine the next formula φ to process and the node n containing φ. The function `satisfy(`n, φ`)` applies the corresponding rule by choosing the appropriate function `satisfy-and(`n, φ`)`, `satisfy-all(`n, φ`)` etc. Most of these functions will add new formulas to n, which is performed by `add-formula(`n, φ`)`. If there are branching points associated with φ, i.e. $D(\varphi, n)$ is not empty, a backup of n is created in the branching point b with $I(b) = \max(D(\varphi, n))$. A possible blocking of n by another node or of another node by n is removed.

For an existential formula $\varphi = \exists \boldsymbol{x}(G(\boldsymbol{c}, \boldsymbol{x}) \wedge \chi(\boldsymbol{c}, \boldsymbol{x}))$, `satisfy-ex(`$n, \varphi$`)` first invokes `blocked(`$n_i$`)` for n and its ancestors to check if n is blocked. For this purpose, the function `equivalent(`n_i, n_j`)` tries to find a mapping π for two nodes

n_i, n_j from $C(n_i)$ to $C(n_j)$ as described in Definition 2. If such a mapping is found, the function `block(`n_i`)` blocks the node n_i and its successors. Otherwise, a new son n_k of n is created with $C(n_k) = C_{\text{old}} \cup C_{\text{new}}$, where $\boldsymbol{d}$ is a vector of new constants for the variables in $\boldsymbol{x}$, C_{old} are the constants in $\boldsymbol{c}$, and C_{new} are the constants in $\boldsymbol{d}$. The guard $G(\boldsymbol{c}, \boldsymbol{d})$ and body $\chi(\boldsymbol{c}, \boldsymbol{d})$ are added to $\Delta(n_k)$, and formulas in $\Delta(n)$ which contain only constants in C_{old} are propagated to $\Delta(n_k)$.

For a disjunction φ, `satisfy-or(`n, φ`)` invokes `choose-alternative(`φ`)` to find the first disjunct ψ to add to n. It then calls `branch(`n, ψ`)`, which creates a new branching point b_{new} and a backup of n in $M(b_{\text{new}})$, and finally adds ψ to n. When a clash occurs, `construct-tableau` calls `backtrack()` to return to the last branching point b_i for which there is another alternative, i.e. $O(b_i)$ is not empty: all nodes n created as a consequence of the last branch are removed by `delete-node(`n`)`, and all nodes modified are replaced with their backups by `restore-tree(`$\{n_1, \ldots, n_k\}$`)`. Then the remaining alternative from $O(b_i)$ is added to the corresponding node and removed from $O(b_i)$.

R$\updownarrow$ is not implemented as a separate rule, but is applied implicitly whenever a formula is added to a node: when `add-formula` is invoked for a formula φ and a node n, it calls `propagate(`$\varphi, \{n_1, \ldots, n_i\}$`)`, which checks if the constants in φ are also contained in the neighbours $n_1, \ldots, n_i$ of n and adds φ to the corresponding nodes.

If a clash occurs and backtracking is impossible, i.e there is no branching point containing another alternative, `construct-tableau` returns "φ `is unsatisfiable`"; if `choose-next-formula` finds no more formulas to process, the tree is complete and "φ `is satisfiable`" is returned together with the tableau that was generated.

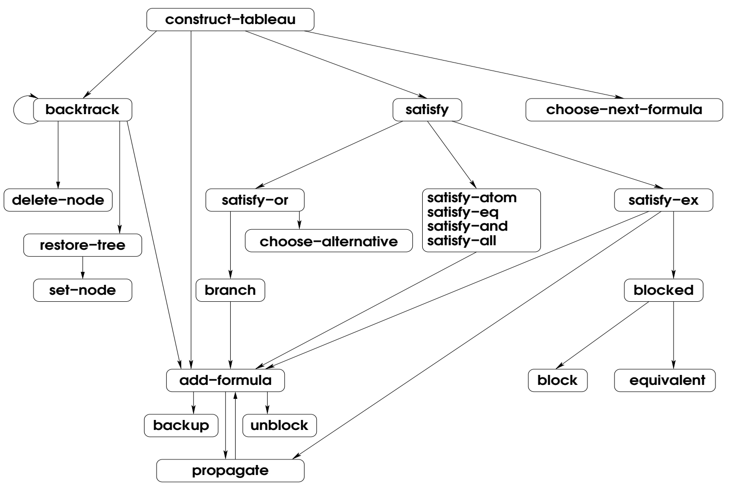


Fig. 1: Function Hierarchy

4 Optimisation

Section 3 describes only a very basic implementation whose performance cannot compete with existing systems for comparably complex logics. To obtain an efficient program, sophisticated optimisation techniques are necessary.

Syntactic Preprocessing. Before SAGA tries to construct a tableau for the input formula φ, it simplifies the syntactic structure of φ to speed up the tableau generation process: obvious tautologies and contradictions are made explicit, a normal form is used which supports their detection by eliminating $\vee$ and $\exists$ and using n-ary conjunctions, and the variables contained in a formula φ are normalised when φ is added to a node. Details can be found in [18] or [19].

Semantic Branching. The naive method to satisfy a disjunction $\varphi \vee \chi$ is to add φ first and, if this causes a clash, add χ afterwards (*Syntactic Branching*). This is rather inefficient because resources have been spent to find out that φ is unsatisfiable in the current tree, but this information is forgotten.

Semantic Branching [7] adds $\neg\varphi \wedge \chi$ to the tree if φ leads to a clash. This makes the information that φ is unsatisfiable explicit and possibly prunes the search space because a tree in which φ is satisfiable is never tested again.

The drawback of semantic branching lies in adding formulas to the tree that are superfluous for a model. If these formulas are complex, they can slow down the tableau generation process because superfluous rule applications take place and superfluous nodes are created. An assessment of the advantages and disadvantages of semantic branching is presented in Section 6.

Backjumping. After a clash, naive backtracking returns to the most recent branching point. Dependency directed backtracking (*backjumping*) [2] instead returns to the most recent branching point *one of the clashing formulas depends on.* Thus, the intermediate BPs, which did not have any influence on the clash, are skipped. The dependency sets described in Section 3 are used to find the right branching point to return to: after a clash between φ and χ in node n, the most recent BP b in $D(\varphi, n) \cup D(\chi, n)$ is determined and the backjump to b is performed.

Boolean Constraint Propagation. Before choosing a disjunct ψ from a disjunction $\varphi \vee \chi$ in a node n and performing a branch for ψ, every disjunct is tested regarding whether it is *closed*, i.e. its negation is already contained in n, or whether it is *open.* If ψ is closed, it is removed from the disjunction (because adding it would lead to an immediate clash), and only the remaining disjuncts are considered for branching (this technique is known as *boolean constraint propagation, BCP* [6]). In particular, if there is only one open disjunct, it is added deterministically to n, and the branch is avoided.

To-do Lists. To efficiently find the next formula to process (i.e. without searching the whole tree constructed so far), a data structure *to-do list* is used. For every kind of formula (atoms, conjunctions, existential restrictions etc.), it contains a list of unblocked nodes which contain un-processed formulas of that kind. These lists are sorted by the node identifiers, which makes it possible to find the "oldest" node containing e.g. an un-processed conjunction in constant time.

5 Heuristics

In this section, the different heuristics implementing the non-deterministic decisions (cf. Section 3) and the different blocking techniques are described.

Branching. The decision which disjunction to choose for the next branch and which one of its disjuncts to test first is crucial for efficiency (cf. Section 6). In SAGA, three different heuristics pursuing different goals are implemented.

MOMS. The heuristic "Maximum Occurrence in disjunctions of Minimum Size" [6] was developed for propositional logic. It considers all clauses (disjunctions) of minimum size and counts the appearances of positive and negative literals (disjuncts). The variable A to branch on is the one with the largest count of A and $\neg A$. If the count for A is larger than the one for $\neg A$, $\neg A$ is tested first and, if this leads to a clash, A is tested afterwards. (MOMS therefore implicitly requires semantic branching.)
The goal is to optimise BCP by increasing the number of closed disjuncts and reaching deterministic expansion as soon as possible. To adapt MOMS for $\mathcal{GF}$, we consider every disjunct appearing in a disjunction of a particular node as a literal and choose the disjunct φ or $\neg\varphi$ for branching in the corresponding way.
One disadvantage of MOMS lies in the fact that it tries the more constrained alternative first, i.e. the alternative that is more likely to fail. This means that it performs nicely for unsatisfiable formulas, but badly for satisfiable ones [18]. Furthermore, it was observed that MOMS interacts adversely with backjumping [19].

iMOMS. Inverted MOMS [18] tries to avoid the disadvantage of being likely to fail with the first alternative by testing φ and $\neg\varphi$ in the opposite order, i.e. it chooses the disjunct which satisfies most of the smallest disjunctions.

Maximise-jump. This heuristic was first used in FaCT [21]. From all disjunctions of a node, it selects the one for which the maximum element in the dependency set is minimal, i.e. the one leading to the furthest backjump. To find the first disjunct to try, FaCT uses a MOMS-style heuristic. In SAGA, the syntactically shortest disjunct is selected because a short formula probably can be tested faster than a long one which is likely to contain existential or universal formulas. Since this approach does not rely on counts of disjuncts like MOMS, we also expect to see the differences in efficiency more clearly.

Choosing the Next Formula. There are two kinds of formulas which are significantly more expensive to process than the remaining ones. Firstly, disjunctions require the creation of a branching point and backups of nodes and, after a clash, all modified nodes have to be restored to their original state. Secondly, for an existential formula in a node n, the blocking test for n has to be performed. Since the blocking condition is defined recursively, each ancestor m_i of n has to be compared with all nodes ℓ_j with $N(\ell_j) < N(m_i)$. For the blocking test itself, all mappings from $C(m_i)$ to $C(\ell_j)$ have to be tested (in the worst case). Therefore, either disjunctions or existential formulas are processed last.

Blocking. The blocking condition for a node n as defined in Section 2 requires $\pi(\Delta(n))$ to be equal to a *restriction* $\Delta(m) \mid_{\pi(C(n))}$ for an ancestor node m. The algorithm also works for an alternate definition of blocking where the same number of constants is required for n and m, i.e. equality of $\Delta(n)$ and $\Delta(m)$ modulo π. This may lead to postponing blocking because the blocking test only succeeds after the creation of some additional nodes, but the test itself becomes significantly more efficient: if the number of constants, atoms, universal formulas etc. is not identical for m and n, it can be aborted immediately without generating a mapping π. In the following, we will refer to the different blocking conditions as *subset-equality blocking* and *equality blocking* respectively.

6 Evaluation of Optimisations and Heuristics

In this section, we present an analysis of the efficiency and the interaction of the heuristics and optimisations described in the previous sections. We used several benchmarks for logics of different expressiveness to see how the heuristics behave for formulas of different complexity: two sets from the "Tableaux 2000 Non-Classical Systems Comparison" (TANCS-2000) [27] benchmark suite and some $\mathcal{GF}$ formulas.

QBF-inv. The "quantified boolean formulas with inverse" benchmark consists of sets of random generated QBF formulas satisfying given parameters, which are translated into the logic $\mathbf{K}^-$ ($\mathbf{K}$ with inverse modality). For this comparison, we used the sets "p-QBF-inv-cnfSSS-K4-C*c*-V4-D4" with $c \in \{10, 20, 30, 40, 50\}$, which are the easiest ones, so that even very inefficient combinations of heuristics can still decide some formulas.

PSAT-inv. The random generated "periodic satisfiability with inverse" formulas are translated into the logic $\mathbf{K}^-$ with global axioms. Again, we used the easiest sets "p-psat-inv-cnf-K4-C*c*-V4-D4" with $c \in \{20, 30, 40, 50\}$.

GFB. Since the QBF and PSAT formulas contain only unary and binary relations, they do not use the complete expressive power of $\mathcal{GF}$. To see how SAGA performs for "proper" $\mathcal{GF}$ formulas, we generated some (simple) "$\mathcal{GF}$ Benchmark" (GFB) formulas. Each set consists of eight random generated formulas with the same width, depth, and maximum arity of the relations.

The QBF benchmark does not require blocking because termination of the algorithm is ensured by the properties of $\mathbf{K}^-$: each constant exists in at most two nodes, and for every grandson n of a node m the maximum modal depth of a formula in $\Delta(n)$ is strictly shorter than it is in $\Delta(m)$. This property makes it possible to regard the blocking test as another heuristic for this benchmark and thus evaluate its efficiency. For the more expressive logics, this is not possible because blocking must be permanently enabled to ensure termination.

The different blocking conditions *equality* and *subset-equality* (cf. Section 5) can only be compared for GFB because in a tree for QBF or PSAT every node contains exactly two constants and the case of subset-equality blocking for a *proper* subset cannot occur.

To evaluate the heuristics by themselves as well as their interactions, we ran every benchmark with every possible combination of heuristics. The figures in the following sections show how many formulas could be solved with the corresponding combination. On the x-axis, the different branching mechanisms are shown: syntactic branching first, then semantic branching with the different branching heuristics. For every combination of the other optimisations, a separate graph is printed. The different measuring points are connected by lines to improve readability.

The benchmarks were run on the following system: Hardware: Pentium-III (733 MHz), 384 MB RAM, 512 MB swap space; Software: Linux (Kernel 2.2), Allegro Common Lisp 6.0.

QBF. Surprisingly, blocking is the most efficient heuristic. With enabled blocking test, up to 26 formulas can be solved, compared to at most 4 without blocking. Although it is very expensive in the worst case, it is obviously still far more efficient than the expansion of the nodes that could be blocked. This indicates that the simple heuristic of comparing the number of constants, atoms, existential formulas etc. before generating a mapping π (cf. Section 5) is sufficient to achieve an efficient blocking test. The speedup could also be explained by regarding the blocking test as a kind of partial model caching (e.g. [21]), which was observed to be very efficient for the TANCS benchmark [13].

Semantic branching and backjumping also provide a significant speedup. While backjumping leads to a rather constant improvement independent of the other optimisations, the speedup delivered by semantic branching is particularly high for efficient combinations of the other heuristics. iMOMS is slightly worse than maximise-jump, and MOMS is far worse than the other branching heuristics. This is true even if backjumping is disabled, i.e. if selecting maximise-jump effectively means choosing a random disjunct, which shows that the main drawback of MOMS is not the interference with backjumping, but the high probability of failing with the first alternative.

Processing R$\exists$ or R$\vee$ first does not have a significant influence, and syntactic simplification has none at all (it is therefore not recorded in the figures). This is probably caused by the structure of the formulas.

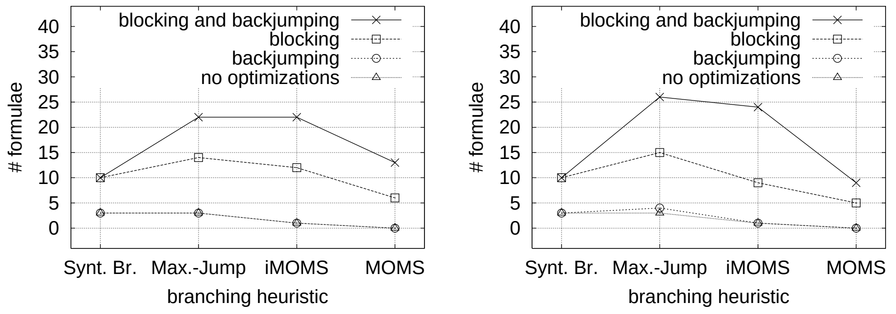


Fig. 2: Heuristics Comparison for QBF with "R∨ Last" (Left) and "R∃ Last" (Right)

PSAT. Semantic branching is by far the most important heuristic, and backjumping is also very efficient. Regarding the optimal rule sequence, we can observe a significant speedup if R∨ is applied before R∃. An explanation for this behaviour is the blocking test: if it is applied only to propositionally expanded nodes, the probability for blocking is higher.

iMOMS is slightly better than maximise-jump, but the difference is irrelevant for efficient combinations of the other heuristics. The same is true for syntactic simplification: if we have an efficient combination of heuristics, disabling syntactic simplification does not significantly slow down the system. This indicates that, in the presence of semantic branching and backjumping, the efficiency is not affected by minor differences in the branching condition or syntactic redundancy of a formula.

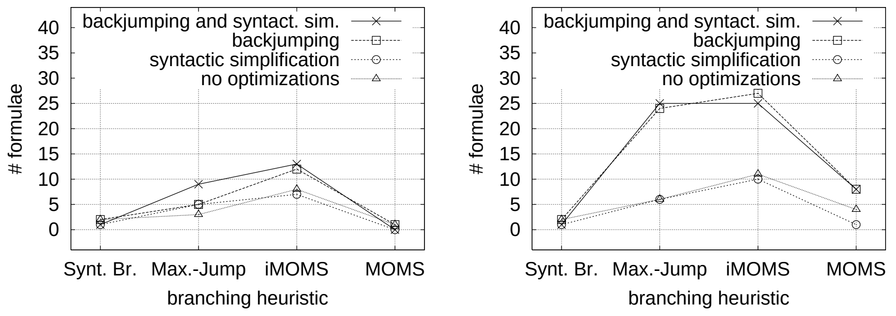


Fig. 3: Heuristics Comparison for PSAT with "R∨ Last" (Left) and "R∃ Last" (Right)

GFB. The results for QBF indicate that blocking, although expensive, results in a speedup. This raises the question if searching for more blocking situations by using subset-equality blocking instead of equality blocking might lead to a further speedup, although the blocking test becomes even more expensive. The measurements show that this is is not the case: subset-equality blocking leads to a higher average time and to fewer solvable formulas. The reason is that in this case, we cannot use the heuristics enabling us to abort the test if a node n is blocked by a node m early if $C(m) > C(n)$, but have to test all subsets of $C(m)$ which have the size of $C(n)$. Syntactic simplification has more influence for GFB than for the other benchmarks, which is probably caused by the simple and random structure of the formulas. The other heuristics behave similarly. (Therefore we do not include a figure.)

Summary. Semantic branching and backjumping deliver a significant speedup for all of our benchmarks. Blocking is useful even when it is not necessary. Among the branching heuristics, maximise-jump and iMOMS perform well, whereas MOMS is slower than choosing a random disjunct. Syntactic simplification does not provide a significant speedup, and the optimal sequence of rule applications differs for the various benchmarks.

7 Comparison with Other Systems

In this section, we examine how well SAGA scales compared to other systems for logics of different complexity, i.e. how fast it can solve formulas from $\mathcal{GF}$ and less expressive logics. We compare our own results with those that were presented in the TANCS-2000 comparison [3] for the QBF-inv/PSAT/PSAT-inv benchmarks.

The comparison systems are DLP [28], a satisfiability tester for PDL [4]; FaCT [20], a tableau algorithm for the description logic $\mathcal{SHIQ}$ [23]; RACE, a TBox and ABox reasoner for the logic $\mathcal{ALCNH}_{R^+}$[16]; and GOST [18], a tableau algorithm for the logic $\mathcal{GF}1^-$ [26], which is a PSPACE-fragment of $\mathcal{GF}$. GOST differs from SAGA in two important aspects: it does not include a blocking test (because it is not necessary to ensure termination for $\mathcal{GF}1^-$), and it uses a simpler backup algorithm: before a branch is performed, the entire tree constructed so far is copied, and during backtracking, the tree is replaced with that copy.

While these systems are tableau algorithms like SAGA and also share most of its optimisations, the last competitor SPASS [30, 31] is a resolution-based first-order theorem prover. MSPASS [24] is a SPASS module translating formulas from the syntax of modal or description logics to first order logic so that their satisfiability can be decided with SPASS.

For this comparison, we used a larger set of the QBF and PSAT formulas than in Section 6. The results for DLP, FaCT, MSPASS and RACE were taken from [3] and those for GOST from [18]. The figures in the following sections show, for every system, how many formulas of each set could be solved.

The SAGA benchmarks and the GF benchmark for SPASS were run on the following system: Hardware: Pentium-III (1 GHz), 512 MB RAM, 1 GB swap space; Software: Linux (Kernel 2.2), Allegro Common Lisp 6.0. Timeout: 600sec (QBF), 1000sec (PSAT), 100sec (GFB).

QBF. Figure 4 shows that Saga is more efficient than Gost and FaCT for most of the sets. While FaCT fails to solve many of the satisfiable formulas from the first sets, (C10/20/30-V4-D4), it performs better for some of the unsatisfiable formulas (C50-V4-D6). It seems that FaCT prunes the search space more efficiently, but needs more time to collect the necessary information.

The comparison with Gost again shows that the blocking test has a positive impact on performance. Furthermore, the more sophisticated backup strategy, though slower for very easy formulas, pays for complex ones: Saga never aborts because of memory exhaustion, whereas this is often the case for Gost. Obviously, this behaviour is a result of the more space-consuming backup strategy.

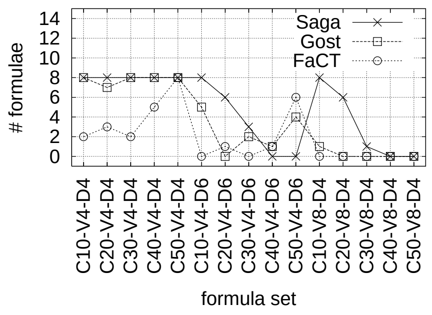
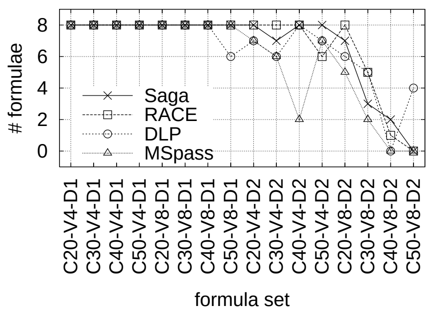


Fig. 4: System Comparison for QBF

PSAT. In order to compare Saga with as many systems as possible, we ran the the PSAT as well as the PSAT-inv benchmark. Although the results presented in this section were produced on different hardware with different timeouts, the differences between the systems are relatively small. The PSAT formulas with depth 1 were easy for all of the systems. For the harder formulas with depth 2, the differences become visible and it turns out that Saga is slightly slower than RACE and slightly faster than DLP and MSpass. For PSAT-inv, Saga is faster than FaCT and similar to MSpass. Again, the difference to FaCT is particularly large for satisfiable formulas.

GFB. Figure 6 shows for every set characterised by the width (w) and depth (d) of the formulas and by the maximum arity (r) of the relations how many formulas could be solved. Among the comparison systems, Spass is the only



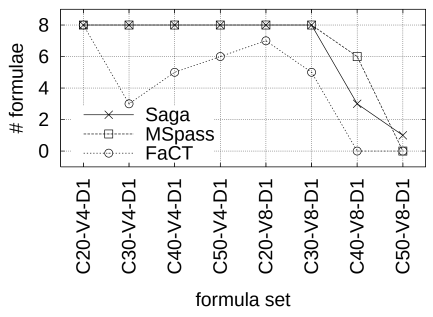


Fig. 5: System Comparison for PSAT (Left) and PSAT-inv (Right)

one that can decide $\mathcal{GF}$ formulas, but, as mentioned before, its characteristics are quite different from SAGA's. Therefore, it is no surprise that the results are also different: while SAGA handles a large depth (d16) of a formula well, it has problems with relations of a higher arity (r4). SPASS shows the opposite behaviour. The sum of decidable formulas is similar.

Though SAGA works only for a fragment of FO and is not significantly faster than SPASS, the benefit of using a tableau algorithm is having a *decision procedure*, i.e. termination is guaranteed (even if it may consume exponential time).

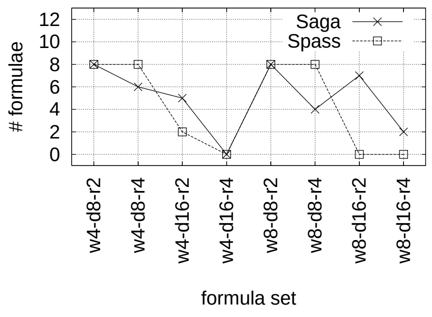


Fig. 6: System Comparison for GFB

8 Conclusion

In this paper, we presented an implementation and empirical analysis of a $\mathcal{GF}$ tableau algorithm. It turned out that, in spite of its worst-case complexity of NEXPTIME (for bounded arity), it performs well for existing benchmark formulas and the $\mathcal{GF}$ formulas we generated. Compared to other systems, Saga's performance is slightly better than FaCT and similar to DLP, RACE an Spass/MSpass. Sophisticated optimisations, in particular backjumping and semantic branching, are necessary to achieve this result. The blocking test, even when it is not necessary to ensure termination, significantly speeds up the program. However, this depends on the heuristics implemented to abort the test early.

The performance analysis presented in this paper is based on random generated formulas, most of which belong to a small fragment of $\mathcal{GF}$. This enables us to compare Saga with several existing systems, but it also means that it may not be representative for real-life problems. The behaviour for realistic knowledge bases is subject to further study.

Acknowledgments

The author of this paper is supported by the DFG, Project No. GR 1324/3-3.

References

[1] H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[2] A. B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.* PhD thesis, University of Oregon, 1995.

[3] R. Dyckhoff, editor. *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2000)*, volume 1847 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 2000.

[4] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Science*, 18:194–211, 1979.

[5] E. Franconi, G. De Giacomo, R. M. MacGregor, W. Nutt, and C. A. Welty, editors. *Proceedings of the International Workshop on Description Logics*, Povo - Trento, Italy, 1998. CEUR.

[6] J. W. Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, 1995.

[7] J. W. Freeman. Hard random 3-SAT problems and the Davis-Putnam procedure. *Artificial Intelligence*, 81:183–198, 1996.

[8] H. Ganzinger, editor. *Proceedings of the 16th International Conference on Automated Deduction (CADE-99)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, Berlin, 1999. Springer-Verlag.

[9] H. Ganzinger and H. de Nivelle. A superposition decision procedure for the guarded fragment with equality. In *14th IEEE Symposium on Logic in Computer Science (LICS)*, Trento, Italy, 1999. IEEE Computer Society Press.

[10] E. Grädel. Guarded fragments of first-order logic: a perspective for new description logics? In Franconi et al. [5]. Extended abstract.

[11] E. Grädel. Decision procedures for guarded logics. In Ganzinger [8].

[12] E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64:1719–1742, 1999.

[13] V. Haarslev and R. Möller. Consistency testing: The RACE experience. In Dyckhoff [3].

[14] V. Haarslev and R. Möller. Description of the RACER system and ist applications. In D.L. McGuinness, P.F. Patel-Schneider, C. Goble, and R. Möller, editors, *Proceedings of the 2001 International Workshop on Description Logics (DL2001)*, volume 49, Stanford, CA, USA, 2001. CEUR.

[15] V. Haarslev and R. Möller. High performance reasoning with very large knowledge bases: A practical case study. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Aritificial Intelligence*. Morgan Kaufman Publishers, San Francisco, USA, 2001.

[16] V. Haarslev, R. Möller, and A.-Y. Turhan. RACE user's guide and reference manual version 1.1. Technical Report FBI-HH-M-289/99, University of Hamburg, CS department, 1999.

[17] C. Hirsch and S. Tobies. A tableau algorithm for the clique guarded fragment. In F. Wolter, H. Wansing, M. de Rijke, and M. Zakharyaschev, editors, *Advances in Modal Logics*, volume 3, Stanford, 2001. CSLI Publications.

[18] J. Hladik. Implementing the n-ary description logic $\mathcal{GF}1^-$. In F. Baader and U. Sattler, editors, *Proceedings of the International Workshop on Description Logics*, Aachen, Germany, 2000. CEUR.

[19] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.

[20] I. Horrocks. Using an expressive description logic: FaCT or fiction? In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*. Morgan Kaufmann Publishers, San Francisco, California, 1998.

[21] I. Horrocks and P. F. Patel-Schneider. Optimising description logic subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.

[22] I. Horrocks and U. Sattler. A description logic with transitive and inverse roles and role hierarchies. *Journal of Logic and Computation*, 9(3):385–410, 1999.

[23] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of the Sixth International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Artificial Intelligence, pages 161–180. Springer-Verlag, 1999.

[24] U. Hustadt, R. A. Schmidt, and C. Weidenbach. MSPASS: Subsumption testing with SPASS. In Lambrix et al. [25].

[25] P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors. *Proceedings of the International Workshop on Description Logics*, Linköping, Sweden, 1999. CEUR.

[26] C. Lutz, U. Sattler, and S. Tobies. A suggestion for an n-ary description logic. In Lambrix et al. [25].

[27] F. Massacci and F. M. Donini. Design and results of TANCS-2000. In Dyckhoff [3].

[28] P. F. Patel-Schneider. DLP system description. In Franconi et al. [5].

[29] V. R. Pratt. Models of program logics. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, 1979.

[30] C. Weidenbach. SPASS: Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 27. Elsevier, 1999.

[31] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topić. System description: SPASS version 1.0.0. In Ganzinger [8].