# Reducing Nondeterministic Tree Automata by Adding Transitions

Ricardo Manuel de Oliveira Almeida
University of Edinburgh, United Kingdom

We introduce saturation of nondeterministic tree automata, a technique that adds new transitions to an automaton while preserving its language. We implemented our algorithm on `minotaut` - a module of the tree automata library `libvata` that reduces the size of automata by merging states and removing superfluous transitions - and we show how saturation can make subsequent merge and transition-removal operations more effective. Thus we obtain a Ptime algorithm that reduces the size of tree automata even more than before. Additionally, we explore how `minotaut` alone can play an important role when performing hard operations like complementation, allowing to both obtain smaller complement automata and lower computation times. We then show how saturation can extend this contribution even further. We tested our algorithms on a large collection of automata from applications of `libvata` in shape analysis, and on different classes of randomly generated automata.

## 1 Introduction

Tree automata are a generalization of word automata to non-linear words (i.e., trees) [10]. They have many applications in model checking [3, 7], term rewriting [11] and related areas of formal software verification, e.g., shape analysis [13]. Several software packages for manipulating tree automata have been developed, e.g., Timbuk [4], Autowrite [11] and `libvata` [16] (on which other verification tools, like Forester [17], are based).

For nondeterministic automata, many questions about their languages are computationally hard. The language universality, equivalence and inclusion problems are PSPACE-complete for word automata and EXPTIME-complete for tree automata [10]. A common approach to solving many instances of the inclusion problem is via the computation of different notions of simulation preorders that at the same time under-approximate language inclusion and are computable in polynomial time [12, 1]. These simulation preorders thus offer a trade-off between computability and expressiveness. Efficient reduction algorithms have been presented both for word automata [8] and for tree automata [2, 6], where language inclusion is witnessed by the membership of a pair of states in a simulation preorder. In our paper, we focus on Heavy(x,y) [6], a polynomial-time algorithm for reducing tree automata, in the sense of obtaining a smaller automaton with the same language, though not necessarily with the absolute minimal number of states possible (in general, as with word automata, there is no unique nondeterministic automaton with the minimal possible number of states for a given language). Heavy(x,y) is based on an intricate combination of transition pruning and state quotienting techniques for tree automata, extending previous work on the words case [8]. Transition pruning is based on the notion that certain transitions may be removed from the automaton because 'better' ones remain. The notion of 'better' is given by comparing the states at the endpoints of the two transitions w.r.t. suitable simulation preorders. The Heavy(x,y) algorithm yields substantially smaller and sparser (i.e., using fewer transitions per state and per symbol) automata than all previously known reduction techniques, and it is still fast enough to handle large instances.

We start by optimizing the computation of simulation preorders in Heavy(x,y). This is done by identifying re-computations that can be skipped, which yields generally faster computation times. We

J. Bouda et al. (Eds.): 11th Doctoral Workshop
on Mathematical and Engineering Methods
in Computer Science (MEMICS 2016)
EPTCS 233, 2016, pp. 33–51, doi:10.4204/EPTCS.233.4

then introduce the dual notion of transition pruning, in which transitions are added to the automaton if 'better' ones exist already. This technique is known as transition saturation and it was previously defined for word automata [9]. As in transition pruning, this technique compares the source states of the two transitions w.r.t. a simulation $R_s$ on the states space, and the target states of the transitions w.r.t. a simulation $R_t$. If saturating an automaton with $R_s$ and $R_t$ preserves the language, we say that $S(R_s,R_t)$ is good for saturation. We provide a summary of all $S(R_s,R_t)$ we found to be or not to be good for saturation.

The motivation behind saturation is that it may allow for new merging of states and transition removal which were not possible by using Heavy alone. Thus saturating an automaton which has been reduced with Heavy(x,y) and then reducing it again might result in an even smaller automaton. We perform an experimental evaluation to measure how much smaller, on average, automata become by interleaving reduction methods with transition saturation. Our results indicate that generally one obtains automata with fewer states, but on some cases with more transitions, than the ones obtained by Heavy(x,y) alone.

In general, one wishes to reduce automata in order to make them more efficient to handle in subsequent computations. Thus, we present a second experimental evaluation showing that the complement automata are much smaller and faster to compute when the automata have previously been reduced with the techniques described above.

We implemented our algorithm as an extension of `minotaut` (source code available [5]), a module of the tree automata library `libvata` [16] where the Heavy algorithm is provided. The experiments described above were performed on a large collection of automata from applications of `libvata` in shape analysis, as well as on different classes of randomly generated tree automata.

## 2 Preliminaries

**Trees and tree automata.** A *ranked alphabet* $\Sigma$ is a set of symbols together with a function $\# : \Sigma \to \mathbb{N}_0$. For $\sigma \in \Sigma$, $\#(\sigma)$ is called the *rank* of $a$. We define a *node* as a sequence in $\mathbb{N}^*$. For a node $v \in \mathbb{N}^*$, we define the $i$-th child of $v$ to be the node $vi$, for some $i \in \mathbb{N}$.

Given a ranked alphabet $\Sigma$, a finite *tree* over $\Sigma$ is defined as a partial mapping $t : \mathbb{N}^* \to \Sigma$ such that for all $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, if $vi \in dom(t)$ then **(1)** $v \in dom(t)$, and **(2)** $\#(t(v)) \geq i$. Note that the number of children of a node $v$ may be smaller than $\#(t(v))$. In this case we say that the node is *open*. Nodes which have exactly $\#(t(v))$ children are called *closed*. Nodes which do not have any children are called *leaves*. A tree is closed if all its nodes are closed, otherwise it is open. By $\mathbb{C}(\Sigma)$ we denote the set of all closed trees over $\Sigma$ and by $\mathbb{T}(\Sigma)$ the set of all trees over $\Sigma$.

A finite nondeterministic *top-down tree automaton* (TDTA) is a quadruple $A = (\Sigma, Q, \delta, I)$ where $Q$ is a finite set of states, $I \subseteq Q$ is a set of initial states, $\Sigma$ is a ranked alphabet, and $\delta \subseteq Q \times \Sigma \times Q^+$ is the transition relation. A TDTA has an unique final state, which we represent by $\psi$. The transition rules satisfy that if $\langle q, \sigma, \psi \rangle \in \delta$ then $\#(\sigma) = 0$, and if $\langle q, \sigma, q_1 \dots q_n \rangle \in \delta$ (with $n > 0$) then $\#(\sigma) = n$. Informally, a run of $A$ reads an input tree top-down from the root, branching into sub-runs on subtrees as specified by the applied transition rules, and it accepts it if every branch ends in $\psi$. Formally, a run of $A$ over a tree $t \in \mathbb{T}(\Sigma)$ (or a $t$-run in $A$) is a partial mapping $\pi : \mathbb{N}^* \to Q$ such that $v \in dom(\pi)$ iff either $v \in dom(t)$ or $v = v'i$ where $v' \in dom(t)$ and $i \leq \#(t(v'))$. Further, for every $v \in dom(t)$, there exists either **a)** a rule $\langle q, a, \psi \rangle$ such that $q = \pi(v)$ and $\sigma = t(v)$, or **b)** a rule $\langle q, \sigma, q_1 \dots q_n \rangle$ such that $q = \pi(v)$, $\sigma = t(v)$, and $q_i = \pi(vi)$ for each $i : 1 \leq i \leq \#(\sigma)$. A *leaf of a run* $\pi$ on $t$ is a node $v \in dom(\pi)$ such that $vi \in dom(\pi)$ for no $i \in \mathbb{N}$.

We write $t \xRightarrow{\pi} q$ to denote that $\pi$ is a $t$-run of $A$ such that $\pi(\varepsilon) = q$. A run $\pi$ is accepting if $t \xRightarrow{\pi} q \in I$.

The *downward language of a state q* in $A$ is defined by $D_A(q) = \{t \in \mathbb{C}(\Sigma) \mid t \xrightarrow{\pi} q, \text{for some run } \pi\}$, while the *language* of $A$ is defined by $L(A) = \bigcup_{q \in I} D_A(q)$. We sometimes write simply $A$ to refer to its language.

**Downward and upward relations.** The behaviour of states in TDTA can be compared by semantic preorders (and their induced equivalences), based on the upward- or downward behaviour of the automaton from these states.

Ordinary downward simulation on tree automata can be characterized by a game between two players, Spoiler and Duplicator. Given a pair of states $(q,r)$, Spoiler wants to show that $(q,r)$ is not contained in the simulation preorder relation, while Duplicator has the opposite goal. Starting in the initial configuration $(q,r)$, Spoiler chooses a transition $q \xrightarrow{\sigma} \langle q_1 \dots q_n \rangle$, where $n = \#(\sigma)$, and Duplicator must imitate it *stepwise* by choosing a transition with the same symbol $r \xrightarrow{\sigma} \langle r_1 \dots r_n \rangle$. This yields $n$ new configurations $(q_1,r_1),\dots,(q_n,r_n)$ from which the game continues independently. If a player ever cannot make a move then the other player wins. Duplicator wins every infinite game. Simulation holds iff Duplicator wins.

A tree branches as one goes downward, but 'joins in' side branches as one goes upward. Therefore a comparison of the upward behaviour of states depends also on the joining side branches as one goes upward in the tree. Thus upward simulation is only defined *relative* to a given other relation $R$ that compares the downward behaviour of states 'joining in' from the sides [1]. One speaks, e.g., of upward simulation *of R*. Thus in the ordinary upward simulation game, starting in the initial configuration $(q,r)$, Spoiler chooses a transition $q' \xrightarrow{\sigma} \langle q_1 \dots q_n \rangle$, where $q = q_i$ for some $i$ and $n = \#(\sigma)$, and Duplicator must imitate it *stepwise* by choosing a transition with the same symbol $r' \xrightarrow{\sigma} \langle r_1 \dots r_n \rangle$, where $r = r_i$, and such that 1) $q_j R r_j$, for every $j \neq i$, and 2) $q \in I \Longrightarrow r \in I$. The game continues from the configuration $(q',r')$, and Spoiler wins if Duplicator ever cannot respond to a move, otherwise Duplicator wins.

While in ordinary downward simulation (resp., upward simulation w.r.t. $R$) Duplicator only knows Spoiler's very next step, in downward $k$-lookahead simulation (resp., upward $k$-lookahead simulation w.r.t. $R$) Duplicator knows Spoiler's next $k$ steps in advance (unless Spoiler's move ends in a deadlocked state - i.e., a state with no transitions). In the case where Duplicator knows *all* steps of Spoiler in the entire downward simulation game in advance (i.e., $k = \infty$), we talk of downward trace/language inclusion (resp., upward trace inclusion w.r.t. $R$). As the parameter $k$ increases, the $k$-lookahead simulation relation becomes larger and thus approximates the respective trace inclusion relation better and better.

The downward/upward $k$-lookahead simulation preorder (denoted $\preceq^{k\text{-}\mathsf{dw}}/\preceq^{k\text{-}\mathsf{up}}(R)$, or just $\sqsubseteq^{\mathsf{dw}}/\sqsubseteq^{\mathsf{up}}(R)$ in the ordinary case) is the set of all pairs $(p,q)$ for which Duplicator has a winning strategy in the respective game. For the downward/upward trace inclusion preorder we write $\subseteq^{\mathsf{dw}}/\subseteq^{\mathsf{up}}(R)$.

Downward/upward $k$-lookahead simulation is PTIME-computable for every fixed $k$ and a good under-approximation of the respective trace inclusion (which is EXPTIME-complete in the downward case [10], and PSPACE-complete for $R = id$ in the upward case).

**Transition pruning and state quotienting.** Given a TDTA $A = (\Sigma, Q, \delta, I)$, certain transitions may be pruned without changing the language, because 'better' ones remain. Given a strict partial order $P \subseteq \delta \times \delta$ on the set of transitions, the pruned automaton is defined as $Prune(A,P) = (\Sigma, Q, \delta', I)$ where $\delta' = \{(p,\sigma,r) \in \delta \mid \nexists (p',\sigma,r') \in \delta.\, (p,\sigma,r)\, P\, (p',\sigma,r')\}$. I.e., if $t\,P\,t'$ then $t$ may be pruned because $t'$ is 'better' than $t$. $Prune(A,P)$ is unique and transitions are removed in parallel without re-computing $P$. Trivially, $L(Prune(A,P)) \subseteq L(A)$. If $L(Prune(A,P)) = L(A)$ also holds we say that $P$ is *good for pruning* (GFP).

We obtain GFP relations by comparing the endpoints of transitions over the same symbol $\sigma \in \Sigma$. Given two binary relations $R_\mathsf{u}$ and $R_\mathsf{d}$ on $Q$, we define $P(R_\mathsf{u}, R_\mathsf{d}) = \{(\langle p, \sigma, r_1 \cdots r_n\rangle, \langle p', \sigma, r'_1 \cdots r'_n\rangle) \mid p\, R_\mathsf{u}\, p'$ and $(r_1 \cdots r_n)\, \hat{R}_\mathsf{d}\, (r'_1 \cdots r'_n)\}$, where $\hat{R}_\mathsf{d}$ is a suitable lifting of $R_\mathsf{d} \subseteq Q \times Q$ to $\hat{R}_\mathsf{d} \subseteq Q^n \times Q^n$: if $R_\mathsf{d}$ is some strict partial order $<_\mathsf{d}$, then $\hat{R}_\mathsf{d}$ is a binary relation $\hat{<}_\mathsf{d}$ s.t. 1) $\forall_{1\leq i\leq n}.\, r_i \leq_\mathsf{d} r'_i$, and 2) $\exists_{1\leq i\leq n}.\, r_i <_\mathsf{d} r'_i$; if $R_\mathsf{d}$ is a non-strict partial order $\leq_\mathsf{d}$, then only condition 1) applies. The relations $R_\mathsf{u}, R_\mathsf{d}$ are chosen such that $P(R_\mathsf{u}, R_\mathsf{d}) \subseteq \delta \times \delta$ is a strict partial order (i.e., of the two relations $R_\mathsf{u}$ and $R_\mathsf{d}$, one must be a strict partial order) that is GFP; see the algorithm Heavy below.

Another method for reducing the size of automata is state quotienting. Given a suitable equivalence on the set of states, each equivalence class is collapsed into just one state. A preorder $\sqsubseteq$ induces an equivalence relation $\equiv := \sqsubseteq \cap \sqsupseteq$. Given $q \in Q$, $[q]$ denotes its equivalence class w.r.t. $\equiv$. For $P \subseteq Q$, $[P]$ denotes the set of equivalence classes $[P] = \{[p] \mid p \in P\}$. The quotient automaton is defined as $A/\equiv := (\Sigma, [Q], \delta_{A/\equiv}, [I])$, where $\delta_{A/\equiv} = \{\langle [q], \sigma, [q_1] \ldots [q_n]\rangle \mid \langle q, \sigma, q_1 \ldots q_n\rangle \in \delta_A\}$. Trivially, $L(A) \subseteq L(A/\equiv)$. If $L(A) = L(A/\equiv)$ also holds, $\equiv$ is said to be *good for quotienting* (GFQ).

**The Heavy algorithm.** Here we describe Heavy(x,y) [6], a tree automata reduction algorithm based on transition pruning and state quotienting. The parameters $x, y \geq 1$ describe the lookahead for the used downward/upward lookahead simulations, respectively, where larger values yield better reduction but are harder to compute. The algorithm is polynomial for fixed $x, y$, and doubly exponential in $x$ (due to the downward branching of the tree) and single exponential in $y$ otherwise. Let $Op(x, y)$ be the following sequence of operations on tree automata, where *RU* stands for removing useless states (i.e., states that cannot be reached from any initial state or from which no tree can be accepted): *RU*, quotienting with $\preceq^{x\text{-}\mathsf{dw}}$, pruning with $P(id, \prec^{x\text{-}\mathsf{dw}})$, *RU*, quotienting with $\preceq^{y\text{-}\mathsf{up}}(id)$, pruning with $P(\prec^{y\text{-}\mathsf{up}}(id), id)$, pruning with $P(\sqsubset^{\mathsf{up}}(id), \preceq^{x\text{-}\mathsf{dw}})$, *RU*, quotienting with $\preceq^{y\text{-}\mathsf{up}}(id)$, pruning with $P(\preceq^{y\text{-}\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$, *RU*. These operations are language preserving, since the used relations are GFP/GFQ [6].

The algorithm Heavy(1,1) just iterates $Op(1,1)$ until a fixpoint is reached. The general algorithm Heavy(x,y) does not iterate $Op(x, y)$, but uses a double loop: it iterates the sequence Heavy(1,1)$Op(x, y)$ until a fixpoint is reached.

The Heavy algorithm is provided in the `minotaut` library [5], making use of `libvata`'s efficient computation of ordinary simulation (for a description of `minotaut`'s implementation of simulation with larger lookaheads see Section 3). Heavy behaves well in practice, significantly reducing both automata of program verification provenience and randomly generated automata [6].

## 3 Efficient Computation of Lookahead Simulations

We performed some optimizations on the computation of the maximal downward lookahead simulation used in Heavy(x,y). In the following we describe the key aspects of the computation in terms of a game between Spoiler and Duplicator. (Upward simulation is similar but simpler, since the tree branches downward.)

**Fixpoint iteration with incremental moves.** We represent binary relations over $Q$ as boolean matrices of dimension $|Q| \times |Q|$. Starting with a matrix $W$ in which all entries are set to TRUE, the algorithm consists of a downward refinement loop of $W$ that converges to the maximal downward $k$-lookahead simulation. In each iteration of the refinement loop, for each pair $p, q$ where $W[p][q]$ is still TRUE:

- Spoiler tries an attack *atk* consisting of a possible move from $p$ of some depth $d \leq k$. Each such attack is built incrementally, for $d = 1, 2, \ldots, k$, in order to give Duplicator a chance to respond

already to a prefix of $atk$ of depth $< k$.

- Duplicator then attempts to defend against the given attack of depth $d$, by finding a matching move $def$ from $q$ by the same symbols s.t. every leaf-state in $def$ is in relation $W$ with the corresponding state in $atk$. (Duplicator's search is done in depth-first mode.) If successful, Duplicator declares victory against this particular (prefix of an) attack and Spoiler tries a new one, since extending the current one to a higher depth is pointless. If unsuccessful and $d < k$, Spoiler builds an attack of the next depth level $d+1$, by extending $atk$ with one new transition from each of its leaf-states. The extra information might enable Duplicator to find a successful defence then.
- Duplicator fails if he could not defend against an attack $atk$ of the maximal depth, either where $atk$ has depth $d = k$ or $d < k$ but $atk$ cannot be extended any more due to all its leaf-states having no outgoing transitions.
- If Duplicator could defend against every attack (or some prefix of it) by Spoiler then $W[p][q]$ stays true, for now.
- In the worst case, for each Spoiler's attack of depth $d$, Duplicator must search through all defences of depth up-to $d$, but often Duplicator wins sooner.
- Similarly, in the worst case, Spoiler needs to try all possible attacks of depth $k$, but often Duplicator already wins against prefixes of some depth $d < k$.

Since the outcome of a local game depends on the values of $W$, the refinement loop might converge only after several iterations. The reached fixpoint represents a relation that is generally not transitive (for $k > 1$), but its transitive closure is the required maximal downward $k$-lookahead simulation preorder $\preceq^{k\text{-dw}}$.

**An Optimization Based on Pre-Refinement.** Following an approach implemented in `Rabit` [15] for word automata, we under-approximate non-simulation as follows. If there exists a tree of bounded depth $d$ that can be read from state $p$ but not from state $q$, then the pair $(p,q)$ cannot be in $k$-lookahead simulation for any $k$. The pre-refinement step iterates through all pairs $(p,q)$ and sets $W[p][q]$ to `false` if such a tree is found witnessing non-simulation. Our experiments show that, for most automata samples, running a pre-refinement with some modest depth $d$ suffices to speed up the $k$-lookahead downward simulation computation.

We now present an optimization that allows to compute lookahead simulation faster. The idea is that attacks which are *good* (i.e., successful) or *bad* (i.e., unsuccessful) may be remembered to skip unnecessary re-computations.

**Semi-global caching of Spoiler's attacks.** An attack is seen as *good* or *bad* within the scope of the *whole game*. Consider the game configuration $(p_1,q_1)$ in Figure 1. Although $q_1$ can read all trees of depth 3 that $p_1$ can read, there are *good* attacks from $p_2$ both against $q_2$ and against $q_3$. Duplicator will find and store these if, when defending against the attack $ac(e,e)$, he first tries the transition to $q_2$ (which can only read $d$), or when defending against $ad(e,e)$ he first tries the transition to $q_3$ (which can only read $c$). After trying possibly all attempts, Duplicator is able to defend against the attack and Spoiler now tries the $b$-transition from $p_1$ to $p_2$. However, all possible sub-attacks are now the same, which makes Duplicator announce defeat on them immediately without any exploration.

In Appendix B two different ways of performing this caching of Spoiler's attacks can be found. The three versions present a trade-off between expressiveness and space required to encode attacks. Our tests

indicate that the semi-global version indeed speeds up the computation on automata with high transition overlaps (i.e., where many states are shared by different transitions).

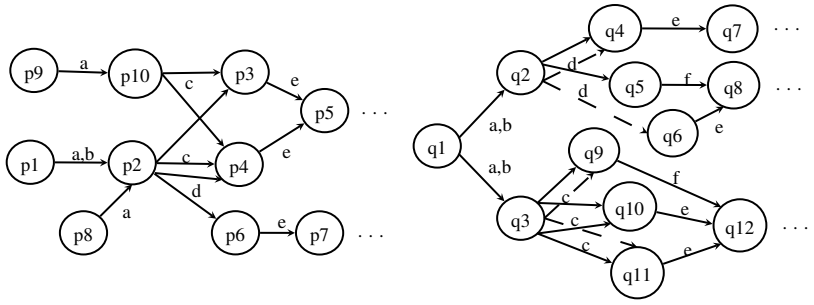


Figure 1: For $W = \{(p_5,q_7),(p_5,q_8),(p_7,q_8),(p_7,q_{12})\}$, all versions of the optimization allow some attacks to be skipped when computing the 3-lookahead downward simulation.

# 4 Saturation of Tree Automata

In Section 2, we described the transition pruning technique, which removes a transition if a 'better' one remains. In this section, we introduce its dual notion, saturation, which adds a transition if a 'better' one exists already. The motivation behind saturation is to pave the way for further reductions when the Heavy algorithm has reached a fixpoint on the automaton (see Section 5). Saturation has been defined for the words case before [9], here we apply it to tree automata.

**Definition 4.1.** *Let $A = (\Sigma, Q, \delta, I)$ be a TDTA, $\Delta = Q \times \Sigma \times Q^+$ and $S \subseteq \Delta \times \Delta$ a reflexive binary relation on $\Delta$. The* S-saturated automaton *is defined as $Sat(A,S) := (\Sigma, Q, \delta_S, I)$, where*

$$\delta_S = \{\langle p', a, q'_1 \dots q'_{\#(a)}\rangle \in \Delta \mid \exists \langle p, a, q_1 \dots q_{\#(a)}\rangle \in \delta \cdot \langle p', a, q'_1 \dots q'_{\#(a)}\rangle \, S \, \langle p, a, q_1 \dots q_{\#(a)}\rangle\}.$$

Since $S$ is reflexive, any transition in the initial automaton is preserved and so $A \subseteq Sat(A,S)$. When the converse inclusion also holds, we say that $S$ is *good for saturation* (GFS). Note that the GFS property is downward closed in the space of reflexive relations, i.e., if $R$ is GFS and $id \subseteq R' \subseteq R$, then $R'$ too is GFS. (or if $R'$ is not GFS, then $R$ too is not GFS).

Given two binary relations $R_s$ and $R_t$ on $Q$, we define $S(R_s,R_t) = \{(\langle p, \sigma, r_1 \cdots r_n\rangle, \langle p', \sigma, r'_1 \cdots r'_n\rangle) \mid pR_sp'$ and $(r_1 \cdots r_n)\hat{R}_t(r'_1 \cdots r'_n)\}$, where $\hat{R}_t$ is the standard lifting of $R_t \subseteq Q \times Q$ to $\hat{R}_t \subseteq Q^n \times Q^n$. Informally, a transition $t'$ is added to the automaton if there exists already a transition $t$ s.t. its source state is $R_s$-larger than the source state of $t'$, and its target states are $\hat{R}_t$-larger than the target states of $t'$. Theorem 1 below proves that $S(\supseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}})$ is GFS. Since the GFS property is downward closed, it follows that $S(\supseteq^{\mathsf{dw}}, \sqsubseteq^{\mathsf{dw}})$, $S(\supseteq^{\mathsf{dw}}, id)$, $S(\sqsupseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}})$, $S(\sqsupseteq^{\mathsf{dw}}, \sqsupseteq^{\mathsf{dw}})$, $S(\sqsupseteq^{\mathsf{dw}}, id)$, $S(id, \subseteq^{\mathsf{dw}})$ and $S(id, \sqsubseteq^{\mathsf{dw}})$ too are GFS. In Theorem 2 (see Appendix A for a proof), we prove that $S(\subseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id))$ is GFS. Thus it follows that $S(\subseteq^{\mathsf{up}}(id), \sqsupseteq^{\mathsf{up}}(id))$, $S(\subseteq^{\mathsf{up}}(id), id)$, $S(\sqsubseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id))$, $S(\sqsubseteq^{\mathsf{up}}(id), \sqsupseteq^{\mathsf{up}}(id))$, $S(\sqsubseteq^{\mathsf{up}}(id), id)$, $S(id, \supseteq^{\mathsf{up}}(id))$ and $S(id, \sqsupseteq^{\mathsf{up}}(id))$ too are GFS.

**Theorem 1.** *$S(\supseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}})$ is GFS.*

*Proof.* Let $A$ be a TDTA and $A_S = Sat(A, S(\supseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}}))$. We will use induction on $n \geq 1$ to show that for every tree $t$ of height $n$ and every run $\pi_S$ of $A_S$ s.t. $t \xrightarrow{\pi_S} p$, for some state $p$, there exists a run $\pi$ of $A$ s.t. $t \xrightarrow{\pi} p$. This shows, in particular, that $A_S \subseteq A$.

In the base case $n = 1$, $t$ is a leaf-node $\sigma$, for some $\sigma \in \Sigma$. Thus for every run $\pi_S$ of $A_S$ such that $t \xrightarrow{\pi_S} p$, for some state $p$, there exists $\langle p, \sigma, \psi \rangle \in \delta_S$. By the definition of $\delta_S$, there exists $\langle q, \sigma, \psi \rangle \in \delta$ s.t. $q \subseteq^{\mathsf{dw}} p$. Consequently, there exists a run $\pi$ in $A$ s.t. $t \xrightarrow{\pi} q$. By $q \subseteq^{\mathsf{dw}} p$, there also exists a run $\pi'$ of $A$ s.t. $t \xrightarrow{\pi'} p$.

For the induction step, let $t$ be a tree of height $n > 1$ and $a$ its root symbol. Thus for every run $\pi_S$ of $A_S$ s.t. $t \xrightarrow{\pi_S} p$, for some state $p$, there exist $\langle p, a, q_1 \dots q_{\#(a)} \rangle \in \delta_S$ and, for each $i : (1 \leq i \leq \#(a))$, a run $\pi_{S_i}$ of $A_S$ s.t. $t_i \xrightarrow{\pi_{S_i}} q_i$. By the definition of $\delta_S$, there exists $\langle p', a, q'_1 \dots q'_{\#(a)} \rangle \in \delta$ s.t. $p' \subseteq^{\mathsf{dw}} p$ and, for every $i : (1 \leq i \leq \#(a))$, $q'_i \supseteq^{\mathsf{dw}} q_i$. Applying the induction hypothesis to each of the subtrees $t_i$, we know that for every $t_i$-run $\pi_{S_i}$ of $A_S$ ending in $q_i$ there is also a $t_i$-run $\pi_i$ of $A$ ending in $q_i$. And since $q'_i \supseteq^{\mathsf{dw}} q_i$ for every $i : (1 \leq i \leq \#(a))$, for each $t_i$ there exists a run $\pi'_i$ of $A$ s.t. $t_i \xrightarrow{\pi'_i} q'_i$. Since there exists $\langle p', a, q'_1 \dots q'_{\#(a)} \rangle \in \delta$, we obtain that there is a run $\pi''$ of $A$ s.t. $t \xrightarrow{\pi''} p'$. From $p' \subseteq^{\mathsf{dw}} p$, it follows that there is also a run $\pi'''$ of $A$ s.t. $t \xrightarrow{\pi'''} p$. □

**Theorem 2.** *$S(\subseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id))$ is GFS.*

The counterexample in Fig. 2 shows that $S(\equiv^{\mathsf{dw}}, \equiv^{\mathsf{up}}(R))$ is not GFS for any relation $R \subseteq Q \times Q$. The remaining counterexamples can be found in Appendix A:

- Figure 8 shows that $S(id, \equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}))$ is not GFS.
- Figure 9 shows that $S(\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}), id)$ is not GFS.
- Figure 10 is inspired by an example for a similar result for linear trees (i.e., words) [9]. It shows that $S(\equiv^{\mathsf{up}}(R), \equiv^{\mathsf{dw}})$ is not GFS for any relation $R \subseteq Q \times Q$.

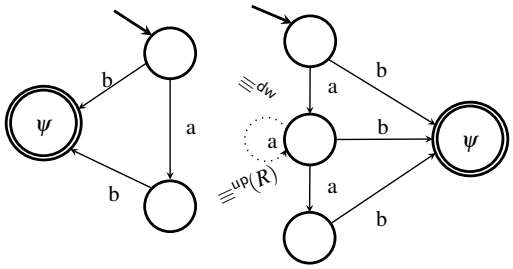


Figure 2: $S(\equiv^{\mathsf{dw}}, \equiv^{\mathsf{up}}(R))$ is not GFS for any relation $R \subseteq Q \times Q$: if we add the dotted transition, the linear tree *aaab* is now accepted. The symbol $b$ has rank 0 and $a$ rank 1.

In Figure 3 we present a table that summarizes these results. The negative results follow from the counterexamples given and the fact that the GFS property is downward closed.

## 5 Experimental Results

As we saw in Section 2, the automaton computed by Heavy corresponds to the local minimum of the sequence of reduction techniques used, i.e., no smaller automaton can be reached by applying that same sequence of steps again. The motivation behind saturation is to change this scenario, since modifying an automaton while preserving its language may leave it in a state where a different local minimum is

| $S$ | $id$ | $\sqsubseteq^{\mathsf{dw}}$ | $\subseteq^{\mathsf{dw}}$ | $\sqsupseteq^{\mathsf{up}}(id)$ | $\supseteq^{\mathsf{up}}(id)$ | $\sqsupseteq^{\mathsf{up}}(\sqsupseteq^{\mathsf{dw}})$ | $\supseteq^{\mathsf{up}}(\supseteq^{\mathsf{dw}})$ |
|---|---|---|---|---|---|---|---|
| $id$ | ✓ | ✓ | ✓ | ✓ | ✓ | × | × |
| $\sqsupseteq^{\mathsf{dw}}$ | ✓ | ✓ | ✓ | × | × | × | × |
| $\supseteq^{\mathsf{dw}}$ | ✓ | ✓ | ✓ | × | × | × | × |
| $\sqsubseteq^{\mathsf{up}}(id)$ | ✓ | × | × | ✓ | ✓ | × | × |
| $\subseteq^{\mathsf{up}}(id)$ | ✓ | × | × | ✓ | ✓ | × | × |
| $\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})$ | × | × | × | × | × | × | × |
| $\subseteq^{\mathsf{up}}(\subseteq^{\mathsf{dw}})$ | × | × | × | × | × | × | × |

Figure 3: GFS relations for tree automata. Relations which are GFS are marked with ✓ and those which are not are marked with ×.

reachable by applying Heavy again. Since saturation adds transitions, in the end an automaton will either have 1) the same number of states and the same or larger number of transitions, 2) the same number of states but fewer transitions, or 3) fewer states. We say that scenarios 2) and 3) correspond to an automaton 'better' than the initial one, and scenario 1) to a 'worse' one.

Our experiments on test automata consisted of first reducing them with Heavy and then alternating between saturation and reduction successively until either a fixpoint is reached or the automata becomes 'worse'. Just like in the case of Heavy, there is no ideal order to apply the saturation/reduction techniques, so we tested multiple possibilities, from which we highlight two versions, Sat1(x,y) and Sat2(x,y), where $x, y \geq 1$ are the lookaheads used for computing $k$-downward and $k$-upward simulations, respectively (see Figure 4). In both Sat1 and Sat2, we chose an order for the operations that ensures that the effect of the saturations is not necessarily cancelled by the reductions immediately after. Intuitively, Sat1 starts by applying both saturations together, in an attempt to obtain a highly dense automaton where more states may be quotiented. Sat2, on the other hand, prevents the automaton from becoming too dense, by interleaving each downward saturation with the upward reductions it may allow. Moreover, each upward reduction not only may allow for new downward saturations to be performed, but it may also have its effect cancelled if the upward saturation is performed immediately after. Thus, in Sat2 downward saturation and upward reductions are iterated in an inner loop before performing any upward saturation. Both versions return the 'best' automaton ever encountered.

We tested the different saturation-based reduction methods on a set of 14,498 automata (57 states and 266 transitions on avg.) from the shape analysis tool Forester [17]. We can see (Figure 5) that, on average, the two versions produced automata containing *both* fewer states and, especially, fewer transitions than Heavy alone. However, this came at the expense of longer running times.

The results that follow focus on the advantage of reducing automata when computing their complement (for which we use `libvata`'s implementation of the difference algorithm [14]). We started by testing on a subset of the Forester sample (Fig. 6 and Fig. 11 in App. C), and we compared direct complementation with reducing automata (with Heavy(1,1) optionally followed by Sat2(1,1)) prior to the complementation and with a final reduction using Heavy(1,1). Due to memory reasons, direct complementation was not feasible for large automata. Thus the sample used is the subset of Forester containing all automata with at most 14 states, in a total of 760 automata. As we can see, all reduction methods yielded significantly smaller complement automata than direct complementation, on average, while running either with similar times or substantially faster. This difference was particularly notorious when the automata were first reduced with both Heavy(1,1) and Sat2(1,1), which, compared to direct complementation, resulted in automata with fewer states (18 vs 27, see Figure 11 in App. C) and fewer transitions

Sat1(x,y)
Loop:
  Sat. w/ $S(\succeq^{x\text{-dw}}, \preceq^{x\text{-dw}})$
  Sat. w/ $S(\preceq^{y\text{-up}}(id), \succeq^{y\text{-up}}(id))$
  Quot. w/ $\preceq^{y\text{-up}}(id)$
  Prune w/ $P(\sqsubset^{\text{up}}(id), \preceq^{x\text{-dw}})$
  Quot. w/ $\preceq^{y\text{-up}}(id)$
  Prune w/ $P(\preceq^{y\text{-up}}(\sqsubseteq^{\text{dw}}), \sqsubset^{\text{dw}})$
  Run Heavy(x,y)

Sat2(x,y)
Loop:
  Loop:
    Sat. w/ $S(\succeq^{x\text{-dw}}, \preceq^{x\text{-dw}})$
    Quot. w/ $\preceq^{y\text{-up}}(id)$
    Prune w/ $P(\prec^{y\text{-up}}(id), id)$
    Prune w/ $P(\sqsubset^{\text{up}}(id), \preceq^{x\text{-dw}})$
    Quot. w/ $\preceq^{y\text{-up}}(id)$
    Prune w/ $P(\preceq^{y\text{-up}}(\sqsubseteq^{\text{dw}}), \sqsubset^{\text{dw}})$
  Sat. w/ $S(\preceq^{y\text{-up}}(id), \succeq^{y\text{-up}}(id))$
  Run Heavy(x,y)

Figure 4: Two saturation-based reduction methods. Both versions return the 'best' automaton ever encountered.

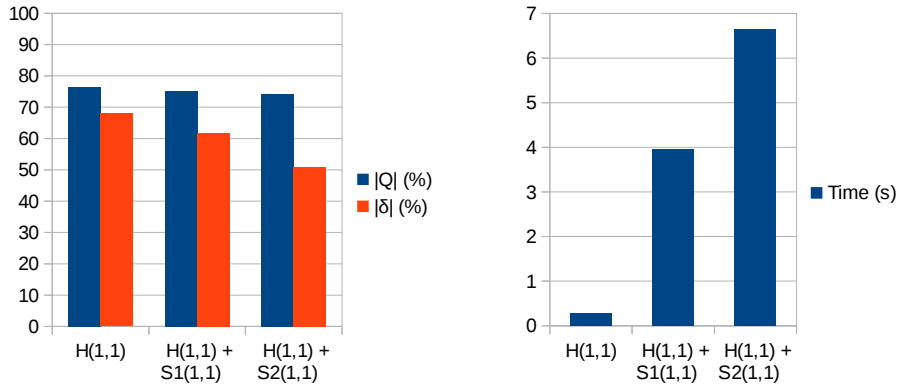


Figure 5: Reduction of Forester automata using saturation methods. The left chart gives the avg. number of states and transitions that remained (in percentage) after application of each method; the right chart compares their running times. Heavy(1,1) followed by Sat2(1,1) reduced the automata the most, but it was also the slowest method.

(649 vs 1750) and at much lower times (0.02s vs 4.86s). Applying Heavy(1,1) in the end reduced the automata even more, with a very low time cost.

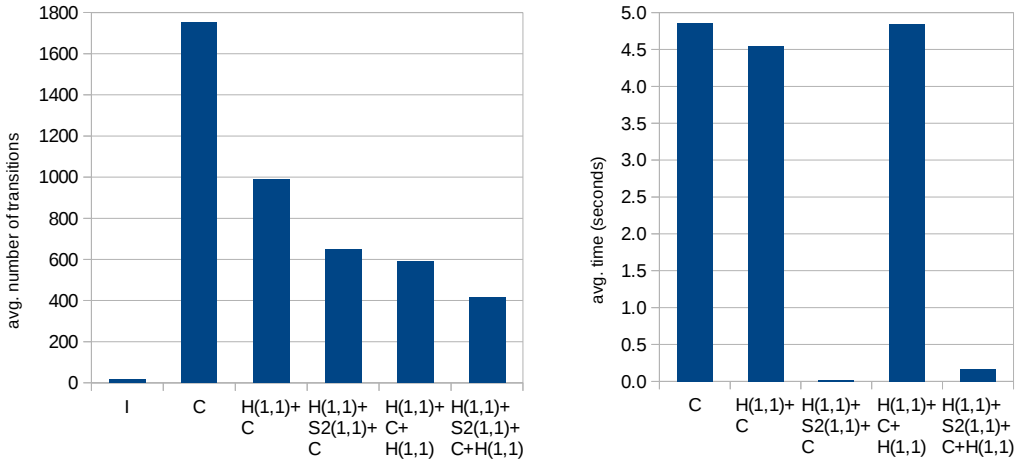


Figure 6: Reducing and complementing Forester automata with at most 14 states. The complement automata have fewer transitions and are faster to compute if the complementation is preceded by applying Heavy(1,1) and Sat2(1,1) - H(1,1)+S2(1,1)+C - or just Heavy(1,1) - H(1,1)+C. Applying Heavy(1,1) in the end reduces even more. We include the initial number of transitions (I) for comparison purposes.

The next experiments were performed on sets of randomly generated tree automata, according to a generalization of the Tabakov-Vardi model of random word automata [18]. Given parameters $n, s$, $td$ (transition density) and $ad$ (acceptance density), it generates tree automata with $n$ states, $s$ symbols (each of rank 2), $n * td$ randomly assigned transitions for each symbol, and $n * ad$ randomly assigned leaf rules. Figure 7 shows the results of complementing automata with $n = 4$ and varying $td$. While the automata tested are very small, for some values of $td$ their complements are quite complex (more than 400 transitions on average). As we can see, applying Heavy not only before but also after the complementation on average yielded significantly smaller automata, especially in terms of transitions, while running with similar times to direct complementation (all average times were below 0.1s). Moreover, the saturation method achieved reductions in the states space which were not possible with Heavy alone. This came at the cost of higher running times and also of returning automata with more transitions - but with still far less transitions then those obtained with direct complementation. Note that for very dense automata ($td \geq 4.0$), the average size of the complement became particularly small. This is because more than half of the automata generated with such $td$ were universal, and thus their complements were empty.

We also tested our algorithms on random automata with 7 states (Figure 13 in App. C), whose complement automata can have, on avg., up to 100 states and more than 30,000 transitions. As above, reducing automata with Heavy both before and after the complementation returned automata with significantly fewer transitions than direct complementation (3,000 vs 35,000 in some cases), but the former was clearly slower (avg. times up to 90s) than the latter (avg. times up to 2.5s) on the automata region where the difference between the two methods was most drastic. Still, for highly dense automata ($td \geq 4$), direct complementation was responsible for the highest times recorded (avg. times between 135s and 2170s). Due to the size of the complement automata, the saturation methods revealed to be too slow to be viable in this case.

All experiments were run on an Intel Core i5 @ 3.20GHz x 4 machine with 8GB of RAM using a

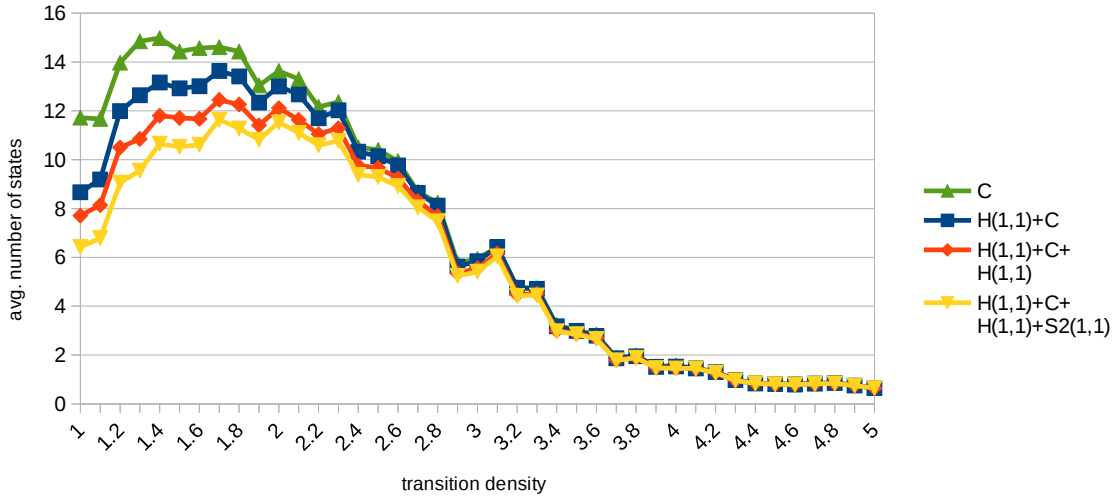


Figure 7: Reducing and complementing Tabakov-Vardi random tree automata with 4 states. Each data point is the average of 300 automata. In general, applying Heavy(1,1) before the complementation (H(1,1)+C) yielded automata with fewer states, on avg., than direct complementation (C). When Heavy(1,1) is also used after the complementation, the difference is even more significant - H(1,1)+C+H(1,1) - and even more when Sat2(1,1) is used - H(1,1)+C+H(1,1)+S2(1,1).

64-bit version of Ubuntu 16.04.

## References

[1] Parosh Aziz Abdulla, Ahmed Bouajjani, Lukás Holík, Lisa Kaati & Tomás Vojnar (2008): *Computing Simulations over Tree Automata*. In: *TACAS*, *LNCS* 4963, pp. 93–108. Available at `http://dx.doi.org/10.1007/978-3-540-78800-3_8`.

[2] Parosh Aziz Abdulla, Lukás Holík, Lisa Kaati & Tomás Vojnar (2009): *A Uniform (Bi-)Simulation-Based Framework for Reducing Tree Automata*. *Electr. Notes Theor. Comput. Sci.* 251, pp. 27–48. Available at `http://dx.doi.org/10.1016/j.entcs.2009.08.026`.

[3] Parosh Aziz Abdulla, Axel Legay, Julien d'Orso & Ahmed Rezine (2006): *Tree Regular Model Checking: A Simulation-Based Approach*. *J. Log. Algebr. Program.* 69(1-2), pp. 93–121. Available at `http://dx.doi.org/10.1016/j.jlap.2006.02.001`.

[4] T. Genet et al. (2015): *Timbuk*. `http://www.irisa.fr/celtique/genet/timbuk/`.

[5] R. Almeida (2016): *minotaut*. `https://github.com/ric-almeida/heavy-minotaut`.

[6] Ricardo Almeida, Lukáš Holík & Richard Mayr (2016): *Reduction of Nondeterministic Tree Automata*, pp. 717–735. Springer Berlin Heidelberg, Berlin, Heidelberg. Available at `http://dx.doi.org/10.1007/978-3-662-49674-9_46`.

[7] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz & Tomás Vojnar (2006): *Abstract Regular Tree Model Checking of Complex Dynamic Data Structures*. In: *SAS*, *LNCS* 4134, pp. 52–70. Available at `http://dx.doi.org/10.1007/11823230_5`.

[8] Lorenzo Clemente & Richard Mayr (2013): *Advanced automata minimization*. In Roberto Giacobazzi & Radhia Cousot, editors: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, ACM, pp. 63–74. Available at `http://doi.acm.org/10.1145/2429069.2429079`.

[9] Lorenzo Clemente & Richard Mayr (2016): *Efficient Reduction of Nondeterministic Automata with Application to Language Inclusion Testing*. Submitted to LMCS.

[10] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison & M. Tommasi (2008): *Tree Automata Techniques and Applications*. Available on: `http://www.grappa.univ-lille3.fr/tata`. Release November, 18th 2008.

[11] I. Durand (2015): *Autowrite*. `http://dept-info.labri.fr/~idurand/autowrite`.

[12] Kousha Etessami (2002): *A Hierarchy of Polynomial-Time Computable Simulations for Automata*. In: *CONCUR*, *LNCS* 2421, pp. 131–144. Available at `http://dx.doi.org/10.1007/3-540-45694-5_10`.

[13] Lukás Holík, Ondrej Lengál, Adam Rogalewicz, Jirí Simácek & Tomás Vojnar (2013): *Fully Automated Shape Analysis Based on Forest Automata*. In: *CAV*, *LNCS* 8044, pp. 740–755. Available at `http://dx.doi.org/10.1007/978-3-642-39799-8_52`.

[14] Haruo Hosoya (2010): *Foundations of XML Processing: The Tree-Automata Approach*, 1st edition. Cambridge University Press, New York, NY, USA. Available at `http://dx.doi.org/10.1017/CBO9780511762093`.

[15] LanguageInclusion.org (Access date:17.12.2015): *RABIT: Ramsey-based Buchi automata inclusion testing*. `http://languageinclusion.org/doku.php?id=tools`.

[16] Ondrej Lengál, Jirí Simácek & Tomás Vojnar (2015): *Libvata: highly optimised non-deterministic finite tree automata library*. `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`.

[17] Ondrej Lengál, Jirí Simácek, Tomás Vojnar, Peter Habermehl, Lukás Holík & Adam Rogalewicz (2015): *Forester: tool for verification of programs with pointers*. `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`.

[18] Deian Tabakov & Moshe Y. Vardi (2007): *Model Checking Buechi Specifications*. In Remco Loos, Szilárd Zsolt Fazekas & Carlos Martín-Vide, editors: *LATA 2007. Proceedings of the 1st International Conference on Language and Automata Theory and Applications.*, Report 35/07, Research Group on Mathematical Linguistics, Universitat Rovira i Virgili, Tarragona, pp. 565–576.

# A Proofs and Counterexamples

For Lemma A.1 and Theorem 2 below we make use of the following auxiliary definitions. For every tree $t \in \mathbb{T}(\Sigma)$ and every $t$-run $\pi$, let $level_i(\pi)$ be the tuple of states that $\pi$ visits at depth $i$ in the tree, read from left to right. Formally, let $(v_1,\ldots,v_n)$, with each $v_j \in \mathbb{N}^i$, be the set of all tree positions of depth $i$ s.t. each $v_j \in dom(\pi)$, in lexicographically increasing order. Then $level_i(\pi) = (\pi(v_1),\ldots,\pi(v_n)) \in Q^n$. We say that $st \in Q^*$ is a subtuple of $level_i(\pi)$, and write $st \leq level_i(\pi)$, if all states in $st$ also appear in $level_i(\pi)$ and in the same order. By lifting preorders on $Q$ to preorders on $Q^n$, we can compare tuples of states w.r.t. $\subseteq^{\mathsf{up}}(id)$.

**Lemma A.1.** *Let $A$ be a TDTA and $(p_1,\ldots,p_n)$ and $(q_1,\ldots,q_n)$ two tuples of states of $A$ such that $(p_1,\ldots,p_n) \subseteq^{\mathsf{up}}(id)(q_1,\ldots,q_n)$. Then, for every $t \in \mathbb{T}(\Sigma)$, every accepting $t$-run $\pi$ and every tuple $(v_1,\ldots,v_n)$ of some leaves of $\pi$ of the same depth $i$ (i.e., $(v_1,\ldots,v_n) \leq level_i(\pi)$) s.t. $(\pi(v_1),\ldots,\pi(v_n)) = (p_1,\ldots,p_n)$, there exists an accepting $t$-run $\pi'$ of $A$ such that $(\pi'(v_1),\ldots,\pi'(v_n)) = (q_1,\ldots,q_n)$ and $\pi'(v) = \pi(v)$ for every leaf $v$ of $\pi'$ other than $v_1,\ldots,v_n$.*

*Proof.* Let $\pi$ be an accepting $t$-run of $A$ s.t. $(\pi(v_1),\ldots,\pi(v_n)) = (p_1,\ldots,p_n)$. We say that an accepting $t$-run $\pi''$ is $i$-good iff i) for every node $v_j$ of $\pi''$, with $j \leq i$, $\pi''(v_j) = q_j$, and ii) for every $v_j$, with $i < j \leq n$, $\pi''(v_j) = p_j$. We will show, by induction on $i$, that for every $i$ there exists an accepting $t$-run $\pi'''$ which is $i$-good and s.t. $\pi'''(v) = \pi(v)$ for every leaf $v$ of $\pi'''$ other than $v_1,\ldots,v_n$. For the particular case of $i = n$ this proves the lemma.

The base case $i = 0$ is trivial, since the accepting $t$-run $\pi$ is 0-good itself.

For the induction step, let $\pi_1$ be an accepting $(i-1)$-good $t$-run of $A$. If $i > n$, the lemma holds trivially. Otherwise, we have $\pi_1(v_i) = p_i \subseteq^{\mathsf{up}}(id)\ q_i$ and thus there exists an accepting $t$-run $\pi_2$ of $A$ s.t. $\pi_2(v_i) = q_i$. And since the upward trace inclusion is parameterized by $id$, it follows, in particular, that for every leaf $v$ other than $v_i$, $\pi_2(v) = \pi_1(v)$. Thus, $\pi_2$ is an accepting $i$-good $t$-run of $A$. Moreover, we have that, on leaves other than $v_1,\ldots,v_n$, the run $\pi_2$ coincides with $\pi_1$ and consequently, by the induction hypothesis, with $\pi$. □

**Theorem 2** $S(\subseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id))$ is GFS.

*Proof.* Let $A$ be a TDTA and $A_S = Sat(A, S(\subseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id)))$. If $\hat{t} \in A_S$, then there exists an accepting $\hat{t}$-run $\hat{\pi}$ of $A_S$. We will show that there exists an accepting $\hat{t}$-run of $A$, which proves $A_S \subseteq A$.

Let us first define an auxiliary notion. For every $t \in \mathbb{T}(\Sigma)$ and every $t$-run $\pi$, we say that $\pi$ is $i$-good iff it does not contain any transition of $\delta_S - \delta$ from any position $v \in \mathbb{N}^*$ s.t. $|v| < i$, i.e., all transitions used in the first $i$ levels of the tree are of $A$.

Next, we will show, by induction on $i$, that for every $i$ there exists an accepting $i$-good $\hat{t}$-run $\hat{\pi}'$ of $A_S$ s.t. $level_i(\hat{\pi}') = level_i(\hat{\pi})$. For $i$ equal to the height of $\hat{t}$, this implies that there exists an accepting $\hat{t}$-run of $A$.

The base case $i = 0$ is trivial, since $\hat{\pi}$ is 0-good itself.

For the induction step, let us first define some auxiliary notions. For every $t \in \mathbb{T}(\Sigma)$ and every $t$-run $\pi$, we say that $level_{i'}(\pi)$ is $j$-good iff $\pi$ does not contain a transition of $\delta_S - \delta$ from a state $\pi(v_k)$, s.t. $k \leq j$ and $\pi(v_k)$ is the $k$-th state of $level_{i'}(\pi)$. We now say that an accepting $\hat{t}$-run $\hat{\pi}''$ of $A_S$ is $(i-1,j)$-good iff i) it is $(i-1)$-good, ii) $level_{i-1}(\hat{\pi}'')$ is $j$-good, and iii) $level_i(\hat{\pi}'') = level_i(\hat{\pi})$.

We will now show, by induction on $j$, that for every $j$ there exists an accepting $(i-1,j)$-good $\hat{t}$-run of $A_S$. Since trees are finitely-branching, we have that for a sufficiently large $j$ there is an accepting $\hat{t}$-run $\hat{\pi}'''$ of $A_S$ which is $i$-good. And since, in particular, $level_i(\hat{\pi}''') = level_i(\hat{\pi})$, this will conclude the outer induction.

For the base case $(i-1,0)$, we know by the hypothesis of the outer induction that there exists an accepting $(i-1)$-good $\hat{t}$-run $\pi_1$ s.t. $level_{i-1}(\pi_1) = level_{i-1}(\hat{\pi})$. Then the $\hat{t}$-run $\pi_2$ which, on the levels below $i$, coincides with $\pi_1$ and, on the levels from $i$ up, coincides with $\hat{\pi}$ too is accepting and $(i-1)$-good. Thus $\pi_2$ is $(i-1,0)$-good.

For the induction step, let $\pi_1$ be an accepting $(i-1,j-1)$-good $\hat{t}$-run of $A_S$, and let $\pi_1'$ be the prefix of $\pi_1$ which only uses transitions of $A$. $\pi_1'$ is thus an accepting run of $A$ over some prefix tree $\hat{t}'$ of $\hat{t}$. Let $v_j$ be the node of $\hat{t}$ s.t. $\pi_1'(v_j)$ is the $j$-th state of $level_{i-1}(\pi_1')$ and $\sigma = \hat{t}(v_j)$ a symbol of rank $r$.

If $r = 0$, then $v_j$ is a leaf of $\hat{t}$ and so there exists a transition $\langle \pi_1'(v_j), \sigma, \psi \rangle$ in $A_S$. By the definition of $\delta_S$, there exists a transition $\langle p, \sigma, \psi \rangle$ in $A$ s.t. $\pi_1'(v_j) \subseteq^{\mathsf{up}}(id)\ p$. Thus there exists an accepting $\hat{t}'$-run $\pi_2$ of $A$ s.t. $\pi_2(v_j) = p$ and for any leaf $v$ of $\pi_2$ other than $v_j$, $\pi_2(v) = \pi_1'(v)$. We now obtain a run over $\hat{t}$ again by extending $\pi_2$ downwards according to $\pi_1$, i.e., $\pi_2(vv') := \pi_1(vv')$, for every leaf $v$ of $\pi_2$ other than $v_j$ and for every $v' \in \mathbb{N}^*$. It follows that $level_i(\pi_2) = level_i(\pi_1) = level_i(\hat{\pi})$. $\pi_2$ is clearly a $(i-1)$-good $\hat{t}$-run of $A_S$ and $level_{i-1}(\pi_2)$ is $j$-good. Thus $\pi_2$ is an accepting $(i-1,j)$-good $\hat{t}$-run of $A_S$.

If $r > 0$, then $v_j$ is not a leaf and so there exists a transition $\langle \pi_1'(v_j), \sigma, \pi_1(v_j1) \dots \pi_1(v_jr) \rangle$ in $A_S$. By the definition of $\delta_S$, there exists a transition $trans$: $\langle p, \sigma, q_1 \dots q_r \rangle$ in $A$ s.t. $\pi_1'(v_j) \subseteq^{\mathsf{up}}(id)p$ and
1) $(q_1 \dots q_r) \subseteq^{\mathsf{up}}(id)(\pi_1(v_j1) \dots \pi_1(v_jr))$. From $\pi_1'(v_j) \subseteq^{\mathsf{up}}(id)\ p$ we have that there exists an accepting $\hat{t}'$-run $\pi_2$ of $A$ s.t. $\pi_2(v_j) = p$ and $\pi_2(v) = \pi_1'(v)$, for every leaf $v$ of $\pi_2$ other than $v_j$. Extending $\pi_2$ with $trans$ we obtain an accepting run of $A$ s.t. $\pi_2(v_jk) := q_k$ for each child $v_jk$ of $v_j$. Applying Lemma A.1 to 1), we obtain that there exists an accepting run $\pi_3$ of $A$ over the same prefix tree of $\hat{t}$ as $\pi_2$ s.t. 2) $\pi_3(v_jk) = \pi_1(v_jk)$ for each child $v_jk$ of $v_j$, and $\pi_3(v) = \pi_2(v) = \pi_1(v)$ for every leaf $v$ of $\pi_3$ other than $v_j1, \dots, v_jr$. We now obtain a run over $\hat{t}$ again by extending $\pi_3$ downwards according to $\pi_1$, i.e., 3) $\pi_3(vv') := \pi_1(vv')$, for every leaf $v$ of $\pi_3$ other than $v_j1, \cdots, v_jr$ and for every $v' \in \mathbb{N}^*$. $\pi_3$ is clearly a $(i-1)$-good $\hat{t}$-run of $A_S$ and $level_{i-1}(\pi_3)$ is $j$-good. From 2) and 3), we obtain that $level_i(\pi_3) = level_i(\pi_1) = level_i(\hat{\pi})$. Thus $\pi_3$ is an accepting $(i-1,j)$-good $\hat{t}$-run of $A_S$. □

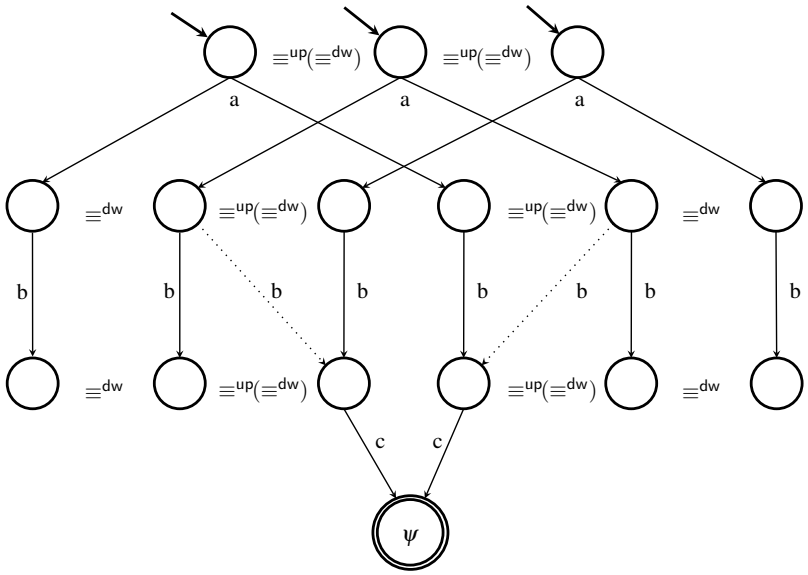


Figure 8: $S(id, \equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}))$ is not GFS: if we add the dotted transitions, the tree $a(b(c), b(c))$ is now accepted. The symbols $c$, $b$ and $a$ have ranks 0, 1 and 2, resp.

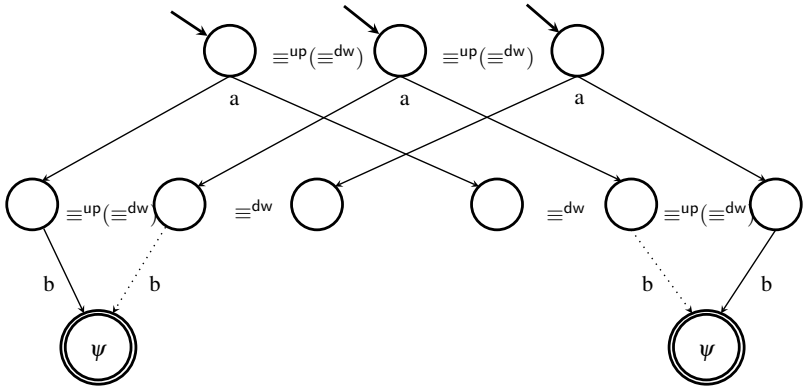


Figure 9: $S(\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}), id)$ is not GFS: if we add the dotted transitions, the tree $a(b,b)$ is now accepted. The symbols $b$ and $a$ have ranks 0 and 1, respectively.

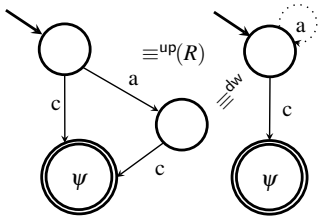


Figure 10: $S(\equiv^{\mathsf{up}}(R), \equiv^{\mathsf{dw}})$ is not GFS for any relation $R \subseteq Q \times Q$ (example adapted from [9]): if we add the dotted transition, the linear tree *aac* is now accepted. The symbol $c$ has rank 0 and $a$ has rank 1.

# B Variants of the Optimization to the Lookahead Simulation Game

In Section 3 we presented an optimization to the computation of the $k$-lookahead downward simulation based on the caching of attacks in the simulation game between Spoiler and Duplicator. In this appendix we present two alternative versions to this optimization, where we change the scope of the attacks cached.

**Local caching of Spoiler's attacks.** Whenever Spoiler uses a transition $t$ in an attack, Duplicator can memorize which states in the automaton are able to defend against the target states of $t$. In Fig. 1 from Section 3, in a round of the simulation game from $(p_2, q_2)$, Spoiler is attempting the attack $d(e, e)$ leading to $p_5, p_7$. Duplicator tries responding with a $d$-transition to $(q_4, q_5)$, and since there is a $e$-transition from $q_4$ to $q_7$ and $p_5 \, W \, q_7$, Duplicator caches the information that, against $q_4$, the first sub-attack is a *bad* one. However, $q_5$ can only read $f$ and so Duplicator will have to try a different defence. Duplicator now tries the $d$-transition leading to $q_4$ and $q_6$ instead. Thanks to the information recorded, Duplicator now only needs to find a defence from $q_6$ against $p_6$, which exists since $q_6$ goes to $q_8$ by $e$ and $p_5 \, W \, q_8$, and so Duplicator declares victory against this particular attack.

Conversely, if the game configuration was $(p_2, q_3)$, after trying to defend against the attack $c(e, e)$ using the $c$-transition to $q_9$ and $q_{10}$, Duplicator could reuse the information that the sub-attack $e$ is *good* against $q_9$ when trying the $c$-transition to $q_9$ and $q_{11}$.

**Global caching of Spoiler's attacks.** Here we expand the scope to the entire $W$-refinement. E.g., the *good* attacks from $p_2$ against $q_2$ or against $q_3$ can be recalled even when a game from a different configuration, say, $(p_8, q_1)$ is played. However, the information about the *bad* attack from, say, $p_3$ against $q_4$ cannot be used outside of the local game in which it was saved, since Duplicator could only defend against it based on the state of $W$ at the time. Note the asymmetry between *good* and *bad* attacks: *good* attacks remain *good* for the rest of the entire computation, but *bad* attacks may become *good* after $W$ changes.

# C More Charts from the Experimental Results

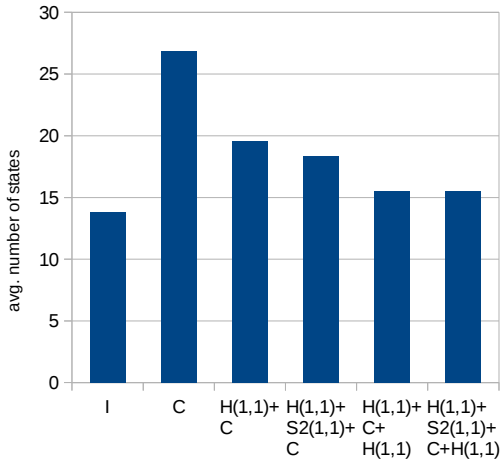


Figure 11: Reducing and complementing Forester automata with at most 14 states. The complement automata have fewer states if the complementation is preceded by applying Heavy(1,1) and Sat2(1,1) - H(1,1)+S2(1,1)+C - or just Heavy(1,1) - H(1,1)+C. Applying Heavy(1,1) in the end reduces even more. We include the initial number of states (I) for comparison purposes.

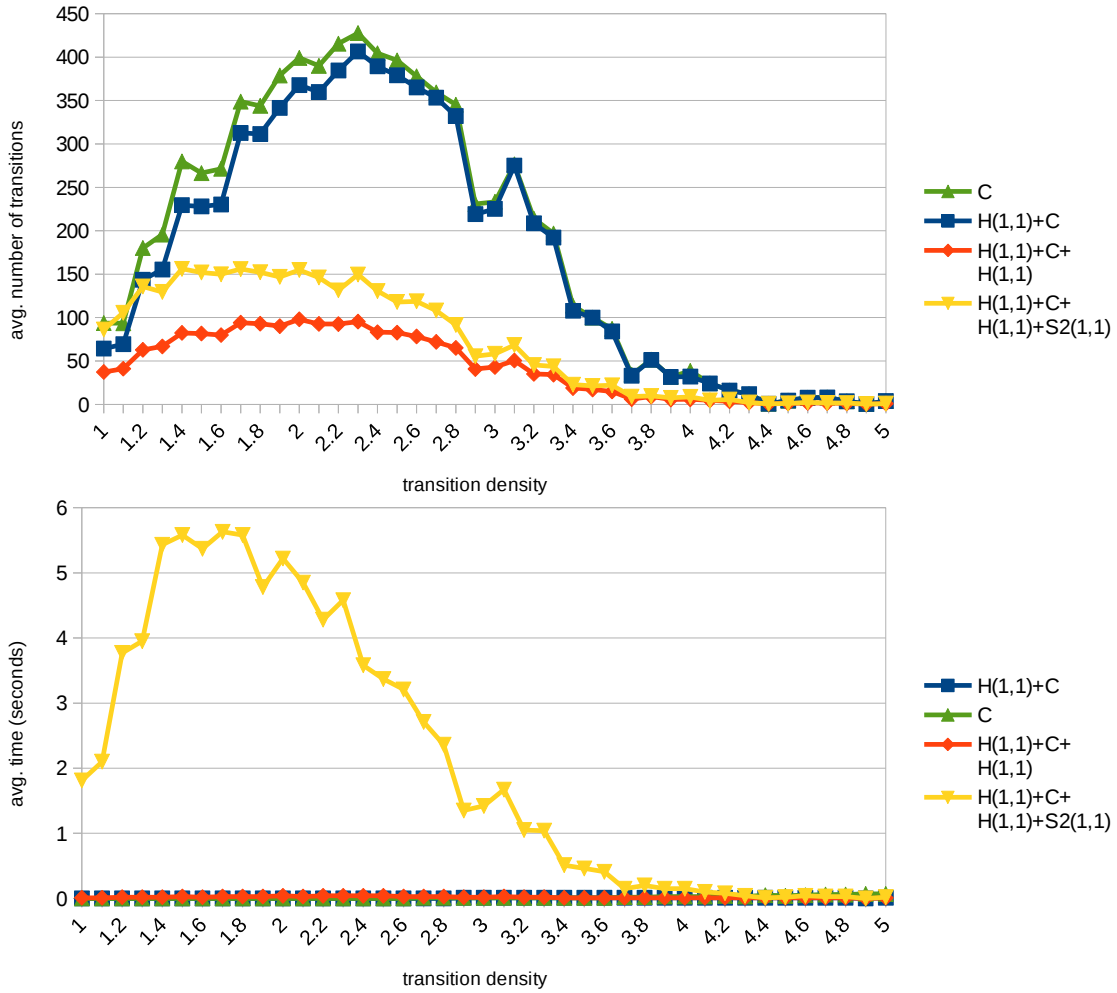


Figure 12: Reducing and complementing Tabakov-Vardi random tree automata with 4 states. Each data point is the average of 300 automata. In general, applying Heavy(1,1) before the complementation (H(1,1)+C) yielded automata with fewer states and transitions, on average, than direct complementation (C). When Heavy is used both before and after the complementation, the difference is even more significant: H(1,1)+C+H(1,1) produced automata with less than $1/3$ of the transitions of C for nearly all values of $td$. Running Heavy followed by Sat2 after the complementation (H(1,1)+C+H(1,1)+S2(1,1)) offered a trade-off between reduction in the states space and in the number of transitions (as well as in time).

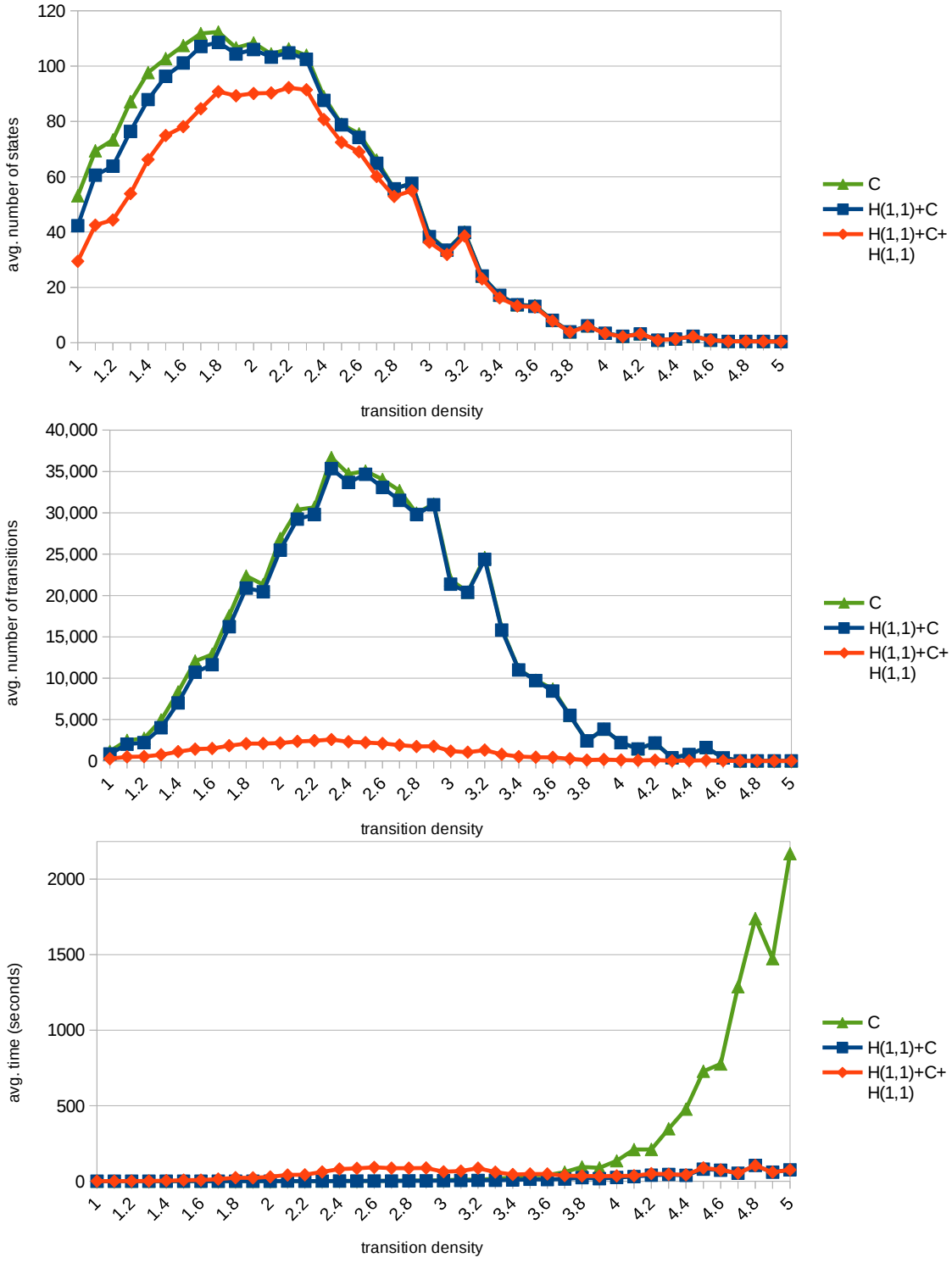


Figure 13: Reducing and complementing Tabakov-Vardi random tree automata with 7 states. Each data point is the average of 300 automata. In general, applying Heavy(1,1) before the complementation (H(1,1)+C) yields smaller automata than direct complementation (C), on average. When Heavy is used both before and after the complementation (H(1,1)+C+H(1,1)), the difference is even more significant: the automata produced by H(1,1)+C+H(1,1) had between 4 and 24 times less transitions than those yielded by C, but the greater reductions took longer to compute. C still took the longest times recorded, for highly dense automata.