Implementing WS1S via Finite Automata

James Glenn[*] Univ. of Maryland

William Gasarch[†] Univ. of Maryland

Abstract

It has long been known that $WS1S$is decidable through the use of finite automata. However, since the worst case running time has been proven to grow extremely quickly, few have explored the implementation of the algorithm. In this paper we describe some of the points of interest that have come up while coding and running the algorithm. These points include the data structures used as wekk as the special properties of the automata, which we can exploit to perform minimization very quickly in certain cases. We also present some data that enable us to gain insight into how the algorithm performs in the average case, both on random inputs ans on inputs that come from the use of Presburger Arithmetic (which can be converted to $WS1S$) in compiler optimization.

1 Introduction

1.1 Definitions

1.1.1 $WS1S$

The language L_{S1S} is the second-order predicate calculus ranging over the natural numbers, with variables $x_1, X_1, x_2, X_2, \ldots$ (to represent numbers and sets of numbers), relation symbols $<$ and $\in$, and the single function symbol s. The relation symbols will be interpreted in the natural way, and the function symbol s will be interpreted as the successor function, with $s^n(x)$ written as $x + n$. Other notable relations can be expressed in this language, for example equality ($x = n$ becomes $\neg(x < n \vee n < x)$, while $X = S$ becomes $(\forall z)(z \in X \iff z \in S)$. We now have atomic formulas of the following forms:

(1) $c_1 < c_2$

(2) $c_1 < x_1 + c_2$

(3) $x_1 + c_1 < c_2$

(4) $x_1 + c_1 < x_2 + c_2$

(5) $c_1 \in S$

(6) $x_1 + c_1 \in X_1$

[*]Dept. of Computer Science, Univ. of MD., College Park, MD 20742, U.S.A. (email: `glennj@cs.umd.edu`).

[†]Dept. of C.S. and Inst. for Adv. Comp. Stud., Univ. of MD., College Park, MD 20742, U.S.A. Supported in part by NSF grants CCR-8803641 and CCR-9020079 (email: `gasarch@cs.umd.edu`).

(7) $x_1 + c_1 \in S$

(8) $x_1 + c_1 \in X_1$

where c_1 and c_2 are natural numbers, x_1 and x_2 are scalar variables, X_1 is a finite set variable, and S is some constant finite set.

Note that formulas of form (1) can be evaluated with no difficulty, and that those of form (5) can be rewritten in terms of $=$ (and hence $<$):

$$x_1 \in \{c_1, \ldots, c_n\} \iff (x_1 = c_1 \vee \cdots \vee x_1 = c_n),$$

so we will not concern ourselves with those forms; we assume, without loss of generality, that formulas are built from atomic formulas of forms (2), (3), (5), (6), (7), and (8). Our procedure then works on sentences such as the following:

$$\begin{gathered}
\exists x \exists y (x < 4 \wedge y < 5) \\
\forall x \forall y (x + 1 < y \rightarrow \exists z (x < z \wedge z < y)) \\
\forall X \exists Y \forall z (\neg(z \in X) \vee (z + 1 \in Y)) \\
\forall x \exists y (y < x) \\
\exists X \exists y (y \in X \wedge \forall z (z \in X \rightarrow z + 1 \in X)) \\
\exists X \forall y (y \in X).
\end{gathered}$$

The set $WS1S$ is the set of sentences of L_{S1S} which are true under the standard interpretation, with quantifiers ranging over natural numbers and finite sets of natural numbers. Hence, of the above sentences, the first three are in $WS1S$ but the last three are not.

1.1.2 *S2S*

The language L_{S2S} is the second-order predicate calculus ranging over words of $\{0,1\}^*$, with variables as for L_{S1S}, relations **prefix**, $<$, and $\in$, and two functions s_0 and s_1. Now $<$ refers to the standard lexicographical ordering of $\{0,1\}^*$, x_1**prefix** x_2 should be read "x_1 is a prefix of x_2", and $\in$ represents set membership as it did in L_{S1S}. The two functions are thought of as concatenation operators, with $s_0(w) = w0$ and $s_1(w) = w1$. $S2S$ is the set of true sentences of L_{S2S}, and is decidable through the use of automata on infinite trees [6].

1.2 Motivation

Hilbert desired a procedure to decide all mathematics. In light of Gödel's Incompleteness Theorem, that is of course impossible. Still, we can ask whether it is reasonable to expect computers to decide those areas of mathematics to which they can be applied. In particular there is a hierarchy (in terms of expressibility) of theories from $WS1S$ to $S2S$ which are decidable through the use of various forms of automata. $S2S$ could have been relevant to descriptive set theory: there is a theorem in that area that can be phrased in L_{S2S}. As it is, humans proved that theorem over a decade before Rabin showed that $S2S$ is decidable, but if the order of events had been different, we could have implemented the decision procedure (which no one has yet done), input the L_{S2S} sentence, and waited for the computer to tell us whether or not it was true. However, we most likely would be waiting for the answer still: even $WS1S$, the weakest theory in the hierarchy, has a provably terrible worst case running time.

Because the decision procedure for $WS1S$ has such a horrible worst-case, study of its implementation has been less than aggressive. Any issues, such as performance or data structure issues, that may be encountered during implementation have been left unknown and undealt with. We have implemented the decision procedure, and in the process we have come across many problems that are worth investigation. One problem that we were forced to deal with is how to keep the size of our automata from growing too fast, and how to do so efficiently. This led us to employ two new strategies for minimization that rely on the structure of the automata we build. One of these methods is a new algorithm that minimizes certain automata in linear time.

We also hope to show whether or not the decision procedure is as slow in practice as the worst case would lead us to believe. One difficulty with this hope is the notion of "in practice". As our "practical" input we shall use both random data and data from real world applications. The latter is possible because we now have several inputs from the use of Presburger Arithmetic by compilers that solve dependency problems to transform parallel loops. It can be shown that any Presburger formula can be converted to a formula in L_{S1S}. If we show that the running time on practical inputs is remotely near the worst case then certainly there is no hope for a practical procedure to decide $S2S$, and even a modified version of Hilbert's program has practically no chance of success.

2 Decidability of $WS1S$

Büchi showed in 1960 that $WS1S$ is decidable through the use of finite automata [1]. However, the cost of the decision procedure can be very high. In fact, it has been proven that for all sufficiently large n, there is a sentence of length n that requires time proportional to $t_{\epsilon \log_2 n}(n)$, where t_k is the tower function with height k [4]. We present a full proof of decidability again in order to indicate where performance issues arise.

The decision procedure for $WS1S$ works by associating with each n-ary formula $f \in L_{S1S}$ a deterministic finite automaton M that operates on an n-track tape. Naturally there will be more than one such automaton, and we will call the set of such automata $DFA(f)$ with the idea being that if $M \in DFA(f)$ then M accepts only those tapes that are solutions to f. To make sense out of that last sentence we must define what we mean by a tape being a solution to a formula. Since formulas work on n-tuples, not n-track tapes, we need a way of translating from one to the other.

The encoding of a set X has a one in place n if $n \in X$ and a zero otherwise. The encoding of a natural number n is n zeros followed by a one. Note that this encoding does not assign meaning to every tape, in particular those with digits after a one. We do, however, wish do give these tapes meaning, and we will do so after the following definitions.

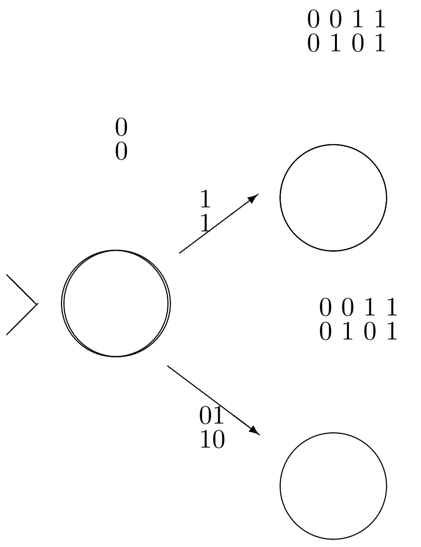
Definition 1 *The empty string is denoted* e*.*

Definition 2 *The symbol on a zero track tape will be denoted* $\perp$*.*

Definition 3 $\mathcal{P}_{finite}(\mathcal{N})$ *is the set of finite subsets of* $\mathcal{N}$*.*

Definition 4 $\Sigma_0 = \{\perp\}$*,* $\Sigma_1 = \{0, 1\}$*, and in general* $\Sigma_n = \{0, 1\}^n$*. These are the alphabets of our automata. We will write these symbols as stacks or column or row vectors as is convenient.*

Definition 5 $Zero(f)$ *is a symbol we can add to the end of a tape without changing its meaning (as far as* f *is concerned). If a track represents a natural number, we can add* 1*'s to it without changing the meaning, and if a track represents a set then we can add* 0*'s to it safely. So if* f *is a formula on triples of* $\mathcal{N}$ *then* $Zero(f) = \substack{1\\1\\1}$*. If* f *has two varibles, the first a set variable and the second a natural number variable, then* $Zero(f) = \substack{0\\1}$*.*



x	0	0	0	1	0	1
y	0	0	0	1	1	0

Figure 1: An automaton in $DFA(x = y)$ and an input it accepts.

Definition 6 *Let γ, our encoding function, which maps $\bigcup_{k=0}^{\infty}(\mathcal{N} \cup \mathcal{P}_{finite}(\mathcal{N}))^k$ to $\bigcup_{k=0}^{\infty} \Sigma_k^*$ be defined by*

(1) $\gamma() = \perp$*;*

(2) $\gamma(\alpha) = 0^{\alpha-1}1$ *if* $\alpha \in \mathcal{N}$*;*

(3) $\gamma(\alpha) = w_0 \cdots w_k$ *where* $w_i = 1$ *if* $i \in \alpha$ *and* $w_i = 0$ *otherwise and* $k = \max_{i\in\alpha} i$ *whenever* $\alpha \in \mathcal{P}_{finite}(\mathcal{N})$*;*

(4) for $(\alpha_1, \ldots, \alpha_n)$, $n \geq 2$ *write* $w = w_1 \ldots w_k$ *for* $\gamma(\alpha_1, \ldots, \alpha_{n-1}$ *and* $u_1 \ldots u_l$ *for* $\gamma(\alpha_n)$*. Extend* w *or* u *with the approprate Zero symbol defined above so they have the same length. Then for* $1 \leq i \leq \max\{k, l\}$ *let* $v_i = \binom{w_i}{u_i}$ *(that is,* v_i *is the element of* Σ_n *that is equal to* w_i *in its top* $n - 1$ *rows and has* u_i *in its bottom row). Then let* $\gamma(\alpha_1, \ldots, \alpha_n) = v_1 \ldots v_{\max\{k,l\}}$*.*

Definition 7 $M \in DFA(f)$ *if and only if*

(1) M *accepts* w *implies* $f(\gamma^{-1}(w))$ *is true; and*

(2) $f(\bar{x})$ *is true implies* M *accepts* $\gamma(\bar{x})$*.*

In the above definition of $DFA(f)$ we must be careful since the encoding function is not onto, and hence its inverse will not be defined for all strings w. We therefore desire an extened inverse function defined over all of $\bigcup_{k=0}^{\infty} \Sigma_k^*$. This desire initially led us to a different encoding for natural numbers. We needed to give meaning the tracks containing no 1's. At first we decided that those tracks would represent zero. This special case led to more complicated automata than we build now. For example, there is a simple formula with n variables that results in an automaton with 2^n states using our original encoding. The current encoding produces a machine with only one state (most improvements are less extreme but still substantial). Preliminary runs over a set of 1000 moderately sized, randomly generated formulas take almost two hours with the old encoding but only about twenty minutes under the current encoding; we speculate that most of the improvement is because the newer encoding leads to less nondeterminism. More motivation for extending the inverse function will be given later.

Definition 8 *Let $u = u_1 \dots u_l \in \{0,1\}^*$. γ_u^{-1} is defined as follows:*

(1) $\gamma_e^{-1}(w) = ()$ for all $w \in \Sigma_0^*$;

(2) $\gamma_0^{-1}(w_1 \dots w_k) = \begin{cases} k & \text{if } w_1 = \dots = w_k = 0 \\ \min\{i-1 \mid w_i \neq 0\} & \text{otherwise} \end{cases}$;

(3) $\gamma_1^{-1}(w_1 \dots w_k) = \{i-1 \mid w_i = 1\}$;

(4) If $w_1 \dots w_k \in \Sigma_n^*$, $n \geq 2$, write $w_i = \begin{pmatrix} w_{i1} \\ \vdots \\ w_{in} \end{pmatrix}$ for $1 \leq i \leq k$ and let $\gamma_u^{-1}(w_1 \dots w_k) = (\gamma_{u_1}^{-1}(w_{11} \dots w_{k1}), \dots, \gamma_{u_n}^{-1}(w_{1n} \dots w_{kn}))$.

Note that we had to define γ^{-1} as a family of functions since γ_0^{-1} and γ_1^{-1} have the same domain but behave differently. This distinction is not important and will be omitted; we will write γ^{-1} whenever we mean one of the γ_u^{-1}'s. Finally, we offer some examples.

$$\begin{aligned}
\gamma(\emptyset) &= e \\
\gamma(3) = \gamma(\{3\}) &= 0001 \\
\gamma(\{4,6\}) &= 0000101 \\
\gamma(3,1,\{1,2\}) &= \begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{matrix} \\
\gamma^{-1}(\perp\perp\perp) &= () \\
\gamma_{00}^{-1}\begin{pmatrix} 0\,0\,0\,0 \\ 0\,0\,0\,0 \end{pmatrix} &= (4,4) \\
\gamma_{10}^{-1}\begin{pmatrix} 0\,0\,0\,0 \\ 0\,0\,0\,0 \end{pmatrix} &= (\emptyset,4) \\
\gamma_{110}^{-1}\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} &= (\{2\},\{0,2,3\},0)
\end{aligned}$$

2.1 Automata Corresponding to Atomic Formulas

We are now ready to describe how to construct an automaton corresponding to any formula f. We will work on a formula f from inside out. That is, we will construct automata for the atomic formulas from which f is built and combine them according to the logical connectives and quantifiers between them.

Let $n, m \in \mathcal{N}$ and consider the formula $f(x)$ be $n < x + m$. We can, without loss of generality, assume that $n \geq m$, since if $n < m$ then f is true for all $x \in \mathcal{N}$; we can easily build an automaton that accepts everything. We can also assume that $m = 0$ since $n < x + m$ is equivalent to $(n - m) < x$. We then build the following (nondeterministic) automaton (Figure 2) that accepts w if and only if $\gamma^{-1}(w) = x$ and $n < x$.

This automaton accepts $0^{n+1}\{0,1\}^*$, and it (or, more properly, its deterministic equivalent) is in $DFA(f)$ since if $x > n$ then $\gamma(x) = 0^x1 = 0^{n+1}0^{x-n+1}1$ which is in the language accepted by the

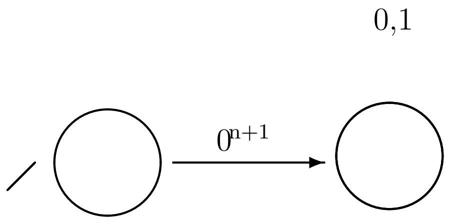


Figure 2: Automaton for $n < x$

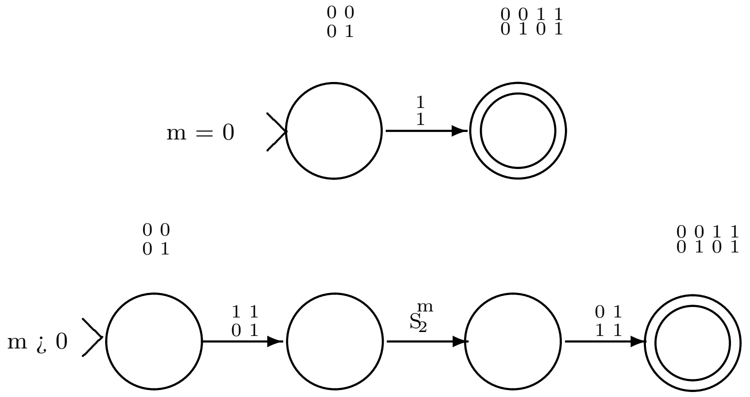


Figure 3: Automata for $y + m \in X$

automaton. Conversely, if w is accepted by the automaton then w must begin with at least $n + 1$ 0's and hence $\gamma^{-1}(w) > n$.

Now let $m \in \mathcal{N}$ and consider the formula $y + m \in X$ which will be denoted $f(X, y)$. We build one of the following finite automata; which one depends on whether m is non-zero (Figure 3). The first of these accepts $\{\genfrac{}{}{0pt}{}{0}{0} \cup \genfrac{}{}{0pt}{}{0}{1}\}^* \genfrac{}{}{0pt}{}{1}{1}\Sigma_2^*$; the second accepts $\{\genfrac{}{}{0pt}{}{0}{0} \cup \genfrac{}{}{0pt}{}{0}{1}\}\{\genfrac{}{}{0pt}{}{1}{0} \cup \genfrac{}{}{0pt}{}{1}{1}\}\Sigma_2^m\{\genfrac{}{}{0pt}{}{1}{0} \cup \genfrac{}{}{0pt}{}{0}{0}\}\Sigma_2^*$.

We could also display automata for atomic formulas of forms (3), (5), (6), (7), and (8).

2.2 Automata for Non-atomic Formulas

2.2.1 Logical Connectives

Constucting an automaton for a formula of the form $f(\bar{x}) = g(\bar{x}) \wedge h(\bar{x})$ or $f(\bar{x}) = g(\bar{x}) \vee h(\bar{x})$ is easy, but does require the use of a data structure that allows us to insert and retrieve ordered pairs quickly. To build the automaton, we first build automata $G \in DFA(g)$ and $H \in DFA(h)$ and combine them in the standard cross-product way to get an automaton F accepting either $L(G) \cap L(H)$ or $L(G) \cup L(H)$. The proof that $F \in DFA(f)$ is simple and will not be done here. Of course, some of the states produced for F may not be reachable; it is desirable when implementing this algorithm to ignore them altogether. By building F forward from (s_G, s_H) (where s_G and s_H are the start states of G and H) and proceeding only to reachable states, we can achieve this goal. This strategy requires that we are able to keep track of what states we have constructed so far; we use an xy-tree for this purpose.

For a formula of the form $f(\bar{x}) = \neg g(\bar{x})$ we can construct $G \in DFA(g)$ and make its final states non-final and vice-versa to get F. As above, the proof that the new automaton is in $DFA(f)$ is simple and not expounded here.

2.2.2 Quantifiers

Finally, we consider formulas involving quantifiers. Let $g(\bar{\alpha}, \alpha_n)$ be an n-ary function and let $h(\bar{\alpha}) = (\exists \alpha_n)(g(\bar{\alpha}, \alpha_n))$ (since we can write $(\forall \alpha_n)(g(\bar{\alpha}, \alpha_n))$ as $\neg(\exists \alpha_n)(\neg g(\bar{\alpha}, \alpha_n))$, we can assume, without loss of generality, that all quantifiers are existential). Let $G = (K, \Sigma_n, \delta, s, F) \in DFA(g)$ and construct

$$H = (K, \Sigma_{n-1}, \Delta, s, F'),$$

where

$$\Delta = \left\{ (q, \sigma, q') \mid \delta\left(q, \begin{smallmatrix} \sigma \\ 0 \end{smallmatrix}\right) = q' \vee \delta\left(q, \begin{smallmatrix} \sigma \\ 1 \end{smallmatrix}\right) = q' \right\}$$

and

$$F' = \left\{ q \in K \mid (\exists w \in Zero(h), u \in \Sigma_1^*, f \in F) \left(\left(q, \begin{smallmatrix} w \\ u \end{smallmatrix}\right) \vdash_G^* (f, e) \right) \right\}$$

(and by $\begin{smallmatrix} \sigma \\ 0 \end{smallmatrix}$ we mean that symbol of Σ_n obtained by writing $\sigma \in \Sigma_{n-1}$ on the first $n-1$ tracks and a zero on the last; by $\begin{smallmatrix} w \\ u \end{smallmatrix}$ we mean that n-track tape with w on the top $n-1$ tracks and u on the last).

Claim 9 *$H \in DFA(h)$.*

Proof. First, suppose $h(\bar{\alpha})$ is true. That means that there exists a α_n such that $g(\bar{\alpha}, \alpha_n)$ is true. Since $G \in DFA(g)$, G must accept $\gamma(\bar{\alpha}, \alpha_n) = \begin{smallmatrix} w_1 \\ u_1 \end{smallmatrix} \begin{smallmatrix} w_2 \\ u_2 \end{smallmatrix} \cdots \begin{smallmatrix} w_n \\ u_n \end{smallmatrix}$. For G to accept that string, there must exist $q_0, \ldots, q_n \in K$ with $q_0 = s$ and $q_n \in F$ such that for all $i, 0 < i \leq n$

$$\left(q_{i-1}, \begin{smallmatrix} w_i \\ u_i \end{smallmatrix} \cdots \begin{smallmatrix} w_n \\ u_n \end{smallmatrix}\right) \vdash_G \left(q_i, \begin{smallmatrix} w_{i+1} \\ u_{i+1} \end{smallmatrix} \cdots \begin{smallmatrix} w_n \\ u_n \end{smallmatrix}\right),$$

from which follows (again, for $0 < i \leq n$)

$$\begin{gathered} \delta\left(q_{i-1}, \begin{smallmatrix} w_i \\ u_i \end{smallmatrix}\right) = q_i, \\ (q_{i-1}, w_i, q_i) \in \Delta, and \\ (q_{i-1}, w_i \ldots w_n) \vdash_H (q_i, w_{i+1} \ldots w_n) \end{gathered}$$

and finally H accepts $w_1 \ldots w_n = w$. But $\gamma(\bar{\alpha})$ will be some prefix of w (remember that w may have been padded to have the same length as u). Let that prefix be $w_1 \ldots w_l$; then we have $w_{l+1}, \ldots, w_n \in Zero(h)$. But now $(q_0, w_1 \ldots w_l) \vdash_H^* (q_l, e)$ and $\left(q_l, \begin{smallmatrix} w_{l+1} \\ u_{l+1} \end{smallmatrix}\right) \vdash_G^* (q_n, e)$ where $w_{l+1} \ldots w_n \in Zero(h)^*$. Therefore $q_l \in F'$ and hence H accepts $w_1 \ldots w_l = \gamma(\bar{\alpha})$.

Now suppose that H accepts $w = w_1 \ldots w_n$. If H accepts that string then we must have, for all $i, 0 < i \leq n$

$$(q_{i-1}, w_i \ldots w_n) \vdash_H (q_i, w_{i+1} \ldots w_n)$$

for some states $q_0, \ldots q_n$ with $q_0 = s$ and $q_n \in F'$. Following this we get, for all $i, 0 < i \leq n$

$$\begin{gathered} (q_{i-1}, w_i, q_i) \in \Delta, \\ (\exists u_i \in \Sigma_1) \left(\delta\left(q_{i+1}, \begin{smallmatrix} w_i \\ u_i \end{smallmatrix}\right) = q_i \right), and \\ \left(q_{i-1}, \begin{smallmatrix} w_i \\ u_i \end{smallmatrix} \cdots \begin{smallmatrix} w_n \\ u_n \end{smallmatrix}\right) \vdash_G \left(q_i, \begin{smallmatrix} w_{i+1} \\ u_{i+1} \end{smallmatrix} \cdots \begin{smallmatrix} w_n \\ u_n \end{smallmatrix}\right). \end{gathered}$$

Further, since $q_n \in F'$ there must exist $y \in Zero(h)$ and $v \in \Sigma_1^*$ such that $\left(q_n, \begin{smallmatrix} y \\ v \end{smallmatrix}\right) \vdash_G^* (f, e)$ for some $f \in F$. Now we have $\left(s, \begin{smallmatrix} w\,y \\ u\,v \end{smallmatrix}\right) \vdash_G^* (f, e)$, so G accepts $\begin{smallmatrix} w\,y \\ u\,v \end{smallmatrix}$. Therefore $g\left(\gamma^{-1}\left(\begin{smallmatrix} w\,y \\ u\,v \end{smallmatrix}\right)\right)$ is true. But $\gamma^{-1}\left(\begin{smallmatrix} w\,y \\ u\,v \end{smallmatrix}\right) = (\gamma^{-1}(wy), \gamma^{-1}(uv))$ and, since $y \in Zero(h)$ $\gamma^{-1}(wy) = \gamma^{-1}(w)$ it follows that

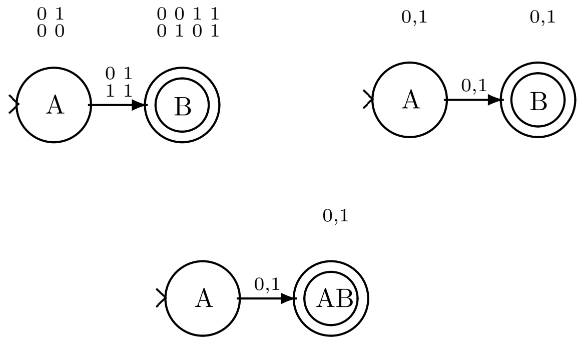


Figure 4: Result of Erasing a Track without Altering Final States

$g(\gamma^{-1}(w), \gamma^{-1}(uv))$ is true. So there exists an α_n, namely $\gamma^{-1}(uv)$, such that $g(\gamma^{-1}(w), \alpha_n)$ is true. In other words, $h(\gamma^{-1}(w))$ is true. □

Two things are worth noting here. First, if the domain of γ^{-1} had not been all of Σ_n^* we would have run into trouble – when we do the nondeterminism modification the new automaton accepts those strings w for which we can add a new track to get a string u accepted by G. What if the new track had no inverse? We would be saying that $f(\alpha_{i_1}, \ldots, \alpha_{i_{n-1}})$ is true because there is an α_{i_n}, namely *garbage*, such that $g(\alpha_{i_1}, \ldots, \alpha_{i_n})$ is true, which is clearly not what we want. We could have addressed this problem by generating a machine that accepts strings w such that you can add a new track b to get a string accepted by G and b has an inverse. We chose to solve the problem by making $domain(\gamma^{-1}) = \Sigma_n^*$ simply because it seemed easier to implement.

Second is the necessity of modifying the set of final states. This is best illustrated by an example (Figure 4). The last automaton rejects e when it should accept all input. Since final state AB is reachable from state A on input 0, we should have made A a final state too. It is easy to check that doing so does indeed result in an automaton that accepts all input.

Now for any sentence $f(\bar{x})$ we can construct the corresponding machine F. It is easy to check whether F accepts $\perp^*$ (recall that $\perp$ is the symbol on tapes with zero tracks). If it does, then $f \in WS1S$, otherwise it is not. However, it may not be easy to build the automaton corresponding to f. For each quantifier in the formula we construct a nondeterministic automaton. Most of the algorithms mentioned so far could be modified to work on nondeterministic automata, except for the algorithm for $\neg f$. There, the input needs to be deterministic for the output to be correct. The algorithm for converting an NFA to a DFA works in worst case $O(2^{|K|})$ time, and hence for each quantifier the size of the machine (and the time needed to create it) may increase exponentially (this, however, should be unlikely).

3 Implementation

3.1 Internal Representation of Automata

If a formula f has n variables, our automata use an alphabet with 2^n symbols. But many of the automata we build for the subformulas of f will not use all n variables. In fact, we will often encounter states that ignore variables that states in the same machine do not. For example, if a machine is driven to state q by an input that has a 1 on some track representing a number, then the transition function for q and all of q's successors cannot depend on the symbol on that track.

We store the complete transition function in an array for each state, which would normally require 2^n entries, but if state q ignores m tracks, then the transition function at q requires only 2^{n-m} units of storage.

While this space saving measure improves running time in some places (for example, if we need to loop over all transitions from a certain state), it makes it more time consuming to compute individual values of the transition function. When the straightforward method is used, it is easy to find $\delta(q, \sigma)$ by simply treating σ as the binary representation of the index into the array holding the transition function. With the above space improvement, we must mask out certain tracks before converting to a number. Since the transition function is evaluated so often, it is critical that this is done as quickly as possible, otherwise the performance of the decision procedure will be degraded greatly. If we do the masking with elementary bitwise operations, we still end up with a number in the range 0 to $2^n - 1$, when what we want is a number in the range 0 to $2^{n-m} - 1$. For example if σ is `10110110` and we are interested in only the second, fourth, fifth, and eighth bits (with the most significant bit numbered one), we mask out the neglected bits to get `x0x10xx0`, which we read as simply `0100`, which is the binary representation of 4, which we use as the index into our array. A slow way to do this follows:

```
result := 0
value := 1
for i := 31 to 0 do
  if bit i of the mask is set then
    if bit i of the symbol is set then
      result = result + value
    end if
    value := value * 2
  end if
end for
```

We do a little precomputation and cut the number of times through the loop to four:

```
result := 0
value := 1
for i := 0 to 3
  x := mask & 0xff                // strip off the last 8 bits in the mask
  y := symbol & 0xff              // do the same for the symbol
  add := table[x][y] * value      // table[x][y] holds precomputed values for
                                  // symbol y and mask x from the algotrithm
                                  // above
  result := result + add
  value := value * 2^bits[x]      // bits[x] = number of bits set in x
  mask := mask / 2^8              // shift mask over to work on the next 8 bits
  symbol := symbol / 2^8          // do the same for symbol
end for
```

Preliminary data shows that the second algorithm allows us to cut the time for our 1000 inputs from 67 minutes to 37 minutes, an improvement of about 45 percent.

3.2 Quantifiers

In our proof of the decidability of $WS1S$, we mentioned the need to recompute the final states of our machines when handling quantifiers. We actually have two options as to how to perform the computation of the new final states. We can do the computation before anything else is done, as implied above, or we can introduce the nondeterminism automaton without changing the final states of the original machine and then carry out the alteration on the resulting machine. (This corresponds to setting

$$F' = \{q \in K \mid (\exists w \in Zero(h), f \in F)((q, w) \vdash_H^* (f, e))\}$$

in the proof above. Little of the remainder of the proof needs correction to show that this is indeed valid.) Our preliminary data show that it is very slightly faster (the improvement is under ten percent) to modify the set of final states before introducing the nondeterminism.

We also have the option of handling n consecutive quantifiers of the same type in one fell swoop rather than in n discrete steps. To do so, however, one must check 2^n values of the transition function of the old machine when computing one value of the transition function at a state in the new machine. We will present data to support the assertion that, on average, it is to our advantage to handle quantifiers in groups (at least for the formulas we have tested so far).

Each nondeterministic machine that is created must be determinized. Profiling has confirmed the common sense feeling that the determinization subroutine is where the algorithm spends most of its time. We compute only those state sets that are reachable from the start state of the non-deterministic machine, hence one of our major concerns is what Wood and Johnson call reachable-state-set maintenance [3]. Within this problem lie at least two subproblems: that of determining if a state set has been seen before, and that of representing the state sets.

3.2.1 Finding a state set

We have implemented two solutions to the former subproblem. State sets can be stored in either a standard binary search tree, or in a tRIe (the latter is a binary tree structure in which the key values are kept only in leaves; the internal structure is such that with each internal node we associate a state of the original NFA, and all of the state sets stored below the left branch of the node will not contain that state, while those under the right branch will). The major problem with the binary search tree is that when we are searching for a particular set, determining which branch to take at a certain node can easily require testing membership of several states. With a tRIe structure, we need only test membership of one state to determine which branch to take. Our initial data show that the tRIe is better by twenty to seventy percent, depending on which structure is used to solve the second subproblem. (It should be noted that in practice we use a hash table to store the state sets and only use the above structures to handle collisions.)

3.2.2 Storing a state set

The state set representation subproblem is essentially that of how to store an m-element subset of $\{0, \ldots, n-1\}$. We have tried several approaches, summarized in the table below.

strcuture	insertion	membership	enumeration	space	time
bit vector	$O(1)$	$O(1)$	$O(n)$	n	9.8
linked list	$O(1)$	$O(m)$	$O(m)$	$64m$	15.2
sorted linked list	$O(m)$	$O(m)$	$O(m)$	$64m$	10.4
binary tree	$O(\log m)$	$O(\log m)$	$O(m)$	$160m$	9.6
hybrid	$O(\log n)$	$O(1)$	$O(m \log n)$	$2n$ to $4n$	11.3 to 11.7
red-black tree	$O(\log m)$	$O(\log m)$	$O(m)$	$193m$	9.9
sorted array	$O(\log m)$	$O(\log m)$	$O(m)$	$32m$	9.7

Here space is measured in bits and times are in minutes and represent running times on our set of 1000 test inputs. Time bounds listed for insertion, membership, and enumeration are generally for the theorectical average case (as opposed to averages we have observed in our study).

The entry labelled “hybrid” actually represents three very similar structures that are the result of an attempt to combine the best aspects of the bit vector structure with the those of the tree structures, and the time bounds listed are very rough. Essentially these structures consist of $\log n$ layers of bit vectors. The bottom layer contains n bits and records for each $i \in \{0, 1, \ldots, n-1\}$ whether i is in the set. The next layer up contains $\frac{n}{2}$ bits recording, for $i \in \{0, 2, \ldots, n-1\}$, if one of i or $i+1$ is in the set. The layers proceed in this manner, with the top layer containing one bit that records if the set is empty or not. The primary reason we constructed these structures was to improve on the slow enumeration of the bit vector structure. However, the constant factors asscociated with the bit vector’s enumeration procedure are apparently so small that we have not seen improved performance on the size automata we have tested so far. We have hope that, on larger automata, these hybrids’ performance will meet or beat that of the bit vector.

Red-black trees are an “approximately balanced” binary tree structure. We decided to implement them because we felt the performance of the standard binary tree structure may have been degraded by the order in which the insertions are performed. We suspected that the insertions frequently were done almost in order, which would result in long, skinny trees and near worst-case (linear) running times. Apparently the overhead that comes with balanced trees negated any gains they could have made.

The sorted array structure takes advantage of the fact that all the insertions are done before either of the other two functions are utilized. In our implementation we first insert elements in constant time into an unsorted array (using a bit vector to avoid inserting duplicates). Once the last insertion is made we can discard the bit vector and sort the array in $O(m \log m)$ time, for an amortized time of $O(\log m)$ time for each of the m insertions. Set membership can then be tested by binary search, and enumeration is simple.

Our test data show the four best performing structures in a statistical tie. Further analysis of our current data, along with test runs on larger machines, may eventually show that some of these structures are clearly better for certain size machines.

3.3 Minimization

Common sense tells us that to keep the running time of our algorithm down, we should periodically minimize our automata. If we keep around machines with twice as many states as necessary, not only will we run out of memory faster, but the and/or algorithm will take four times as long to run. The nondeterminism routine will be even worse, for a machine’s bloat will increase exponentially. Empirical results tell us that minimization is not only desirable, but practically mandatory – it is easy to find inputs that will run out of memory on common workstations. These studies also tell us that we should minimize our automata at almost every opportunity. Because minimizing automata is so important (in addition to being interesting in its own right), we have looked at three ways of approaching the problem. One is well established in the literature, and two are new strategies we developed during the implementation of the algorithm. Of these, one is easy to do but gives only partial results, and the other is an algorithm that exploits an interesing property of certain automata.

3.3.1 Trap State Reduction

Suppose we have a formula of the form $f(\bar{x}) = g(\bar{x}) \wedge h(\bar{x})$ and let q be a state of H that accepts nothing. Then our new automaton F will have a possibly large set of equivalent states $\{(q, q') \mid q' \in G\}$ that accept nothing. If we detect that we have generated a state (q_G, q_H) such that q_G or q_H accepts nothing, we do not have to explore any of the possible successors of that state, and if we generate (q'_G, q'_H) with the same property, we can immediately mark it as equivalent to (q_G, q_H). This strategy will work in the nondeterminism algorithm as well. Empirical data show that though this is helpful, it alone will not solve the problem of large machines.

3.3.2 Algorithms for Minimizing Arbitrary Automata

We also need to employ a general-purpose automaton minimization algorithm. We have tested three candidates: the standard algorithm, one Hopcroft presented in 1976 which works in $O(\mid \Sigma \mid n\log n)$ time in the worst case [2], and Brzozowski's algorithm (reverse, determinize, reverse, determinize).

Brzozowski's algorithm has been championed by some for its excellent performance on small automata despite its poor worst case running time [7]. However, in our tests the exponential nature of that algorithm showed up for even small automta. For example, the automaton we construct for the formula

$$x_2 + 4 \in X_0 \vee (2 < x_2 \wedge x_2 + 4 \notin X_1) \vee (x_2 + 2 \in X_1 \wedge X_1 \subset X_0)$$

has 26 states and takes 2.35 seconds to minimize via reversal to an equivalent machine with 21 states. Hopcroft's algorithm minimizes the same machine in less than a hundredth of a second (the great disparity in times has also been seen when feeding our automata into other automata toolkits such as `grail`).

The difference between the standard and Hopcroft's algorithms is less marked. For smaller machines, the standard algorithm is faster. We have yet to determine exactly where the cutoff is, but it seems to be around 1000 states. Hopcroft's algorithm is hurt by the fact that it needs to compute the inverse transition relation, which is done in $O(\mid \Sigma \mid n)$ time using our internal representation of automata. We need no preprocessing to help the standard algorithm with our current representation.

3.3.3 Layered Automata

Finally, we can exploit the special structure of some of our automata to use a new algorithm that runs in $O(\mid \Sigma \mid n)$ time. The algorithm is presented below. The special requirement is that the automaton has no non-trivial cycles (formally, if $(q_0, w_0) \vdash (q_1, w_1) \vdash \cdots \vdash (q_n, w_n) \vdash (q_0, w_{n+1})$ then $q_0 = q_1 = \cdots = q_n$). All of the automata generated for atomic formulas are of this structure. It is preserved through the operations of negation, and, and or. Unfortunately, existential quantifiers can destroy the structure (Figure 5).

Preliminary tests show that on average, our algorithm works almost twice as fast as Hopcroft's. It is therefore worthwhile to use it whenever possible; that is on subformulas with no quantifiers. If the formula has all its quantifiers in front, this is nearly every subformula.

The algorithm is as follows:

(1) For each state q, initialize $votes_q$ to $\mid \{\sigma \mid \delta(q, \sigma) = q\} \mid$.

(2) Divide those states with $votes_q = \mid \Sigma \mid$ into final and non-final; mark each member of these groups as equivalent to the others, mark them all as level 0 and let $currentlevel = 1$.

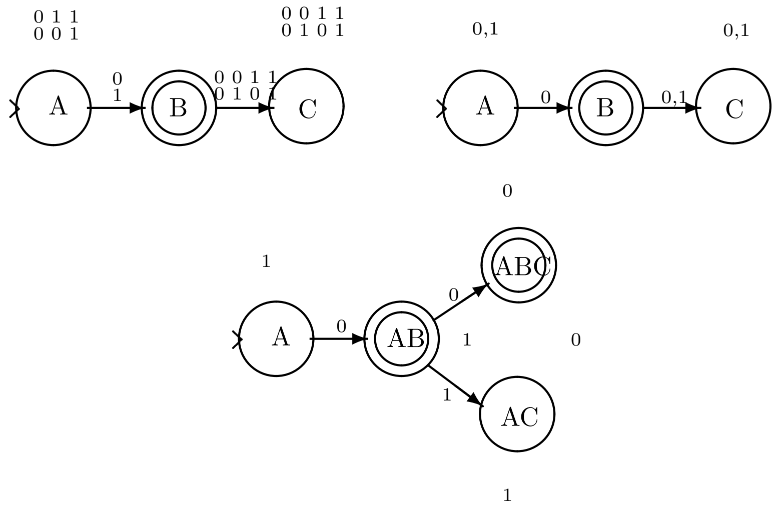


Figure 5: Layers are not Preserved by Nondeterminism

(3) For each state q in level 0 and $\sigma \in \Sigma$, if $\delta(q, \sigma) = q'$ then increment $votes_{q'}$.

(4) Collect all states with $votes_q =| \Sigma |$ and call them *elected*. If *elected* is empty, then halt.

(5) If any state q in *elected* is equivalent to a node in level $currentlevel - 1$ mark them as equivalent, move it from *elected* to level $currentlevel - 1$, and for each $\sigma \in \Sigma$ if $\delta(q, \sigma) = q'$ then increment $votes_{q'}$, adding q' to *elected* if $votes_{q'} =| \Sigma |$.

(6) If any states were added to *elected* by step 5, then repeat it.

(7) Mark those states remaining in *elected* as in level *currentlevel* and check for equivalence among them.

(8) Elect new states into *elected* as in steps 3 and 5, increment *currentlevel*, and return to step 4.

The following lemmas provide the basis of a correctness proof of the above algorithm.

Theorem 10 *Suppose that levels* 0 *through* n *are minimized and* $currentlevel = n + 1$. *Then the following hold before each execution of step* 5.

(a) Any state in elected has at least one transition into level n.

(b) No state in elected has a transition to a different node in elected

(c) No state in elected is equivalent to any node in levels 0 *to* $n - 1$.

(d) If a state q *in elected has* $delta(q, \sigma) = r$ *for some* $\sigma \in \Sigma$ *and* r *in level* n, *and* $s \neq r$ *is also in level* n *then* q *is not equivalent to* s.

(e) A state q *in elected is equivalent to state* r *in level* n *if and only if they are both final or both non-final and*

$$\forall \sigma \in \Sigma((\delta(q, \sigma) \equiv \delta(r, \sigma)) \vee (\delta(q, \sigma) = q \wedge \delta(r, \sigma) \equiv r))$$

Theorem 11 *Suppose that levels* 0 *though* n *are minimized and* $currentlevel = n+1$. *Then before each execution of step* 7, *two nodes* q *and* q' *in elected are equivalent if and only if they are both final or both non-final and*

$$\forall\sigma \in \Sigma((\delta(q,\sigma) \equiv \delta(q',\sigma)) \vee (\delta(q,\sigma) = q \wedge \delta(q',\sigma) = q'))$$

3.4 Performance

We now present the results of running our algorithm on randomly generated formulas. We have several sets of such formulas, each set contains 250 formulas with the same number of clauses and quantifiers, and for which we specified the same maximum constant during the generation process. The average times (in milliseconds) for each set are given below.

quantifiers	2		3		4	
clauses	8	16	8	16	8	16
$max = 2$	21	36	30	53	43	88
$max = 3$	24	43	36	66	60	124
$max = 4$	28	49	48	80	86	186
$max = 5$	31	56	57	101	107	204
$max = 6$	37	66	67	123	143	348
$max = 7$	41	73	89	148	189	485
$max = 8$	x50	88	127	176	230	624

From this data we can see that adding a quantifier increases running time by a factor of about 2. Also, the effect of increasing the size of the constants is much more profound on formulas with more quantifiers.

4 Presburger Arithmetic

Presburger Arithmetic, the first-order theory of the integers with + and <, is decidable. There is an algorithm to determine the truth of a Presburger formula in $2^{2^{2^{pn \log n}}}$ time [5] which works by eliminating quantifiers, converting infinite searches to finite searches. At the end, all the algorithm has to do is check the finite (but very large) number of cases. Another approach to the problem is to convert a Presburger Formula into $WS1S$ and run the algorithm described in the last section on that formula. We do not expect this method to provide a faster algorithm for deciding Presburger Arithmetic since the bound for $WS1S$ is even worse that that for Presburger Arithmetic, but the formulas that result provide us with a source of real-world test inputs (Presburger Formulas are used by compilers that optimize loops for parallel execution). We hope to show that we can verify reasonable Presburger formulas in a reasonable amount of time.

An atomic Presburger formula has the form *term* < *term* where a *term* is either a *constant*, a *variable*, the sum of two *terms*, or a constant multiple of a *term*. We can assume without loss of generality that the multipliers are powers of two (and in fact this is the way in which our implementation treats them).

In the conversion from Presburger Arithmetic to $WS1S$, the integer variables become set variables with a set variable representing an integer if its characteristic function, written out as a string, is the binary representation of the integer, with an additional bit for the sign. This system is not as nice as one in which the two's complement are used (we have two representations of zero, and adding is more difficult), but since we need to encode arbitrarily large integers we would need an infinite number of bits for two's complement, and hence infinite sets which of course we cannot use. The following formulas mean that a set X represents a term (given a free variable z to use):

$$f(X,z,0) = (\forall z)(z \geq 1 \rightarrow z \notin X)$$

$$f(X,z,c) = C_0 \vee \cdots \vee C_n$$

where

$$C_0 = \begin{cases} 0 \in X & \text{if } c < 0 \\ 0 \notin X & \text{if } c > 0 \\ \mathit{true} & \text{otherwise} \end{cases}$$

and

$$C_i = \begin{cases} i \in X & \text{if the } \mathit{ith} \text{ bit of } c \text{ is set} \\ i \notin X & \text{otherwise.} \end{cases}$$

$$\begin{array}{rr} f(X,z_1,-t) = (\forall T)(f(T,z_2,t) \rightarrow (& (\forall z_1)[z_1 \geq 1 \rightarrow (z_1 \in T \iff z_1 \in X)] \quad \wedge \\ & [(0 \in X \iff 0 \in T) \vee f(T,z_1,0)]) \quad) \end{array}$$

$$\begin{array}{rr} f(X,z,2^n t) = (\forall T)(f(T,z,t) \rightarrow (& [(0 \in X \iff 0 \in T) \vee f(T,z,0)] \quad \wedge \\ & (\forall z)(0 < z \leq n \rightarrow z \notin X) \quad \wedge \\ & (\forall z_1)(z > n \rightarrow (z \in T \iff z + n \in X) \quad)) \end{array}$$

$$\begin{array}{rrr} f(X,z,t_1+t_2) = & \forall(T_1,T_2)([f(T_1,z,t_1) \wedge f(T_2,z,t_2)] \rightarrow (& \\ & (0 \notin T_1 \wedge 0 \notin T_2 \wedge 0 \notin X \wedge \mathit{adds}(T_1,T_2,X)) & \vee \\ & (0 \in T_1 \wedge 0 \in T_2 \wedge 0 \in X \wedge \mathit{adds}(T_1,T_2,X)) & \vee \\ & (0 \in T_1 \wedge 0 \notin T_2 \wedge \mathit{bigger}(T_2,T_1) \wedge 0 \notin X \wedge \mathit{subtracts}(T_2,T_1,X)) & \vee \\ & (0 \in T_1 \wedge 0 \notin T_2 \wedge \mathit{bigger}(T_1,T_2) \wedge 0 \in X \wedge \mathit{subtracts}(T_1,T_2,X)) & \vee \\ & (0 \in T_1 \wedge 0 \notin T_2 \wedge \mathit{same}(T_1,T_2) \wedge f(X,z,0)) & \vee \\ & (0 \notin T_1 \wedge 0 \in T_2 \wedge \mathit{bigger}(T_1,T_2) \wedge 0 \notin X \wedge \mathit{subtracts}(T_1,T_2,X)) & \vee \\ & (0 \notin T_1 \wedge 0 \in T_2 \wedge \mathit{bigger}(T_2,T_1) \wedge 0 \in X \wedge \mathit{subtracts}(T_2,T_1,X)) & \vee \\ & (0 \notin T_1 \wedge 0 \in T_2 \wedge \mathit{same}(T_1,T_2) \wedge f(X,z,0)) &)) \end{array}$$

where

$$\begin{array}{rcl} \mathit{bigger}(X,Y) & = & (\exists z_1)(z_1 > 0 \wedge z_1 \in X \wedge z_1 \notin Y \wedge (\forall z_2 > z_1)(z_2 \in X \rightarrow z_1 \in Y)) \\ \mathit{same}(X,Y) & = & (\forall z > 0)(z \in X \iff z \in Y) \\ \mathit{adds}(X,Y,Z) & = & (\exists C)(1 \notin C \wedge (\forall z > 0)\mathit{carry}(X,Y,Z,C,z)) \\ \mathit{subtracts}(X,Y,Z) & = & (\exists B)(1 \notin B \wedge (\forall z > 0)\mathit{borrow}(X,Y,Z,B,z)) \end{array}$$

and finally $\mathit{carry}(X,Y,Z,C,z)$ is a predicate that is true if and only if the carry and sum bits work out at position z. In particular, it is true for

$z \in X$	$z \in Y$	$z \in C$	$z \in Z$	$z+1 \in C$
T	T	T	T	T
T	T	F	F	T
T	F	T	F	T
T	F	F	T	F
F	T	T	F	T
F	T	F	T	F
F	F	T	T	F
F	F	F	F	F

and false for all other cases. The *borrow* predicate is similar.

Last, the formula $t_1 < t_2$ can be converted to a formula involving $f(T_1, z, 0)$, $f(T_2, z, 0)$, and $bigger(T_2, T_1)$.

The upshot of this is that every $+$, $-$, or $*$ in a term will generate at least one quantifier in the $WS1S$ formula. The total number of quantifiers, and hence exponential blowups in the $WS1S$ decision procedure, can get quite large. We can help ourselves out somewhat by storing the machine generated for the formula for $t_1 + t_2$ and reusing it instead of building it from scratch each time. Indeed, the formula above boils down to a nine state DFA. In fact, we can save the machines generated by any size sum. In general such a machine will have $O(2^n)$ states for the sum of n terms.

We have converted a random set of Presburger formulas to L_{WS1S} and run the $WS1S$ decision procedure on them. The random formulas we generated were all of a simple form, so the raw results below will most likely not hold for more general formulas. However, the *trends* shown below may still appear in the general case.

	maximum constant		
quantifiers	4	8	16
2	2.21	3.21	10.85
3	10.45	28.08	
4	68.03		

This table (for which we are still gathering data) gives average times (in seconds) for deciding formulas with a given number of variables and a given bound on the constants that appear in the formula. We can see that changing either parameter has a such profound effect on the running time that filling out this table even a few rows or columns past its current boundary will be quite time consuming.

References

[1] J. R. Büchi. Weak second order arithmetic and finite automata. *Zeitscrift fur mathematische Logic und Grundlagen der Mathematik*, 6:66–92, 1960.

[2] J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computation*, pages 189–196. Academic Press, 1976.

[3] J. Johnson and D. Wood. Instruction computation in subset construction. In *Proceedings of the First International Workshop on Implementing Automata*, pages 1–9, August 1996.

[4] A. R. Meyer. Weak monadic second order theory of succesor is not elementary-recursive. In *Logic Colloquim*, number 453 in Lecture Notes in Mathematics, pages 132–154. Springer-Verlag, 1974.

[5] D. C. Oppen. Elementary bounds for Presburger arithmetic. In *5th ACM Symposium on Theory of Computing*, pages 34–37, 1973.

[6] M. O. Rabin. Decidablilty of second-order theories and automata on infinite trees. *Trans. AMS*, 141:1–35, July 1969.

[7] B. W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Eindhoven University of Technology, 1995.