ABSTRACT

Title of Dissertation: Implementing $WS1S$ via Finite Automata

James R. Glenn, Doctor of Philosophy, 1998

Dissertation directed by: Professor William I. Gasarch
Department of Computer Science

Abstraction is good.

Implementing $WS1S$ via Finite Automata

by

James R. Glenn

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1998

Advisory Committee:

Professor William I. Gasarch, Chairman
Professor Clyde P. Kruskal
Professor Donald R. Perlis
Professor Carl H. Smith
Professor Michael C. Laskowski

© Copyright by

James R. Glenn

1998

TABLE OF CONTENTS

List of Tables **iv**

List of Figures **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Definitions 2

1.2.1 $WS1S$ 2

1.2.2 $S2S$ 5

1.3 Outline 5

2 Decidability of $WS1S$ **7**

2.1 Automata Corresponding to Atomic Formulas 11

2.2 Automata for Non-atomic Formulas 16

2.2.1 Logical Connectives 16

2.2.2 Quantifiers 17

2.3 Testing membership in $WS1S$ 21

3 Implementation **22**

3.1 The Encoding Function and Automata for Atomic Formulas 22

3.2 Internal Representation of Automata 25

3.3 Algorithm for Union and Intersection 28

4 Determinization **29**

4.1 Final States of Our Nondeterministic Automata 29

4.2 Multiple Quantifiers 31

4.3 Maintaining Reachable States 32

4.3.1 Finding a state set 32

4.3.2 Storing a state set 36

4.3.3 Results 37

4.4 Data Structures 39

4.4.1 Hierarchical Bit Vectors 39

4.4.2 8-ary Hierarchical Bit Vectors 55

5 Minimization **58**

5.1 Trap State Reduction 59

5.2 Algorithms for Minimizing Arbitrary DFAs 59

5.3 Layered Automata 60

6 Presburger Arithmetic **67**

7 Conclusion **71**

Bibliography **73**

LIST OF TABLES

LIST OF FIGURES

2.1 An automaton in $DFA(x = y)$ and an input it accepts. 10
2.2 An automaton in $DFA(c < x)$. 12
2.3 An automaton in $DFA(x_1 \in X_2)$ 13
2.4 An automaton in $DFA(x_1 + c \in X_2)$. 14
2.5 An automaton in $DFA(x_1 + c < x_2)$. 15
2.6 An automaton in $DFA(c \in X)$. 15
2.7 Result of erasing a track without altering final states 21

3.1 Another possible machine for $x_1 \in X_2$. 24

4.1 Seven sets stored in a binary search tree. 33
4.2 Seven sets stored in a binary tRIe. 34
4.3 Seven sets stored in an n-ary tRIe. 35
4.4 An HBV representing $\{2, 3, 6\}$ 40
4.5 The path taken when sorting a set stored in an HBV. 42
4.6 Possible paths through the right sub-HBV. 46
4.7 An $8HBV$ representing $\{9, 15, 42, 45, 47\}$. 55

5.1 Layers are not preserved by nondeterminism 62

Chapter 1

Introduction

1.1 Motivation

Hilbert desired a procedure to decide all mathematics [3]. In light of Gödel's Incompleteness Theorem, that is of course impossible [2]. Still, we can ask whether it is reasonable to expect computers to decide some subset of mathematics. 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 [1, 8, 9]. $S2S$ could have been relevant to descriptive set theory: there is a theorem in that area that can be phrased as a query to $S2S$. 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 query to $S2S$, 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 superexponential worst case running time.

Because the decision procedure for $WS1S$ has such a daunting worst-case, study of its implementation has been lacking. Questions of how data structures

and algorithms affect the performance of the decision procedure have been left unasked and therefore unanswered. We have implemented the decision procedure, and in the process we have come across many problems that are worth investigation.

To carry out our investigation, we wish to have some "practical" inputs on which to run the decision procedure. One problem with this is the definition of "practical." As our inputs we will 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} [7].

1.2 Definitions

1.2.1 $WS1S$

The language L_{S1S} is the second-order predicate calculus ranging over the natural numbers, with variables $x_1, X_1, x_2, X_2, \ldots$ (with the x_i representing numbers and the X_i representing sets of numbers; if we need to refer to a variable and we do not know or do not care what its domain is we will write χ_i), 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) $c_1 < x_i + c_2$

(3) $x_i + c_1 < c_2$

(4) $x_i + c_1 < x_j + c_2$

(5) $c_1 \in S$

(6) $x_i + c_1 \in X_k$

(7) $x_i + c_1 \in S$

(8) $c_1 \in X_k$

where c_1 and c_2 are natural numbers, x_i and x_j are scalar variables, X_k is a finite set variable, and S is some constant finite set.

Note that formulas of form (1) and (5) can be evaluated with no difficulty, and that those of form (7) can be rewritten in terms of $=$ (and hence $<$):

$$x_i + c_0 \in \{c_1, \dots, c_n\} \iff (x_i + c_0 = c_1 \vee \dots \vee x_i + c_0 = c_n).$$

Formulas of form (2) can be written as $(c_1 - c_2) < x_i$ which is interesting only when $c1 - c2 \geq 0$ (otherwise it is true). Formulas of form (3) can be rewritten $\neg(c_2 \leq x_i + c_1)$ which is equivalent to $\neg(c_2 - c_1 \leq x_i)$ and also to $\neg(c_2 - c_1 - 1 < x_i)$.

Finally, formulas of form (4) can be rewritten as either $x_i + (c_1 - c_2) < x_j$ when $(c_1 \geq c_2)$, or $x_i < x_j + (c_2 - c_1)$ otherwise. The latter is equivalent to $\neg(x_j + (c_2 - c_1) \leq x_i)$, which is the same as $\neg(x_j + (c_2 - c_1 - 1) < x_i)$.

Therefore, we can assume, without loss of generality, that our formulas are built from atomic formulas of the following forms (where $c \geq 0$):

(a) $c < x_i$;

(b) $x_i + c < x_j$;

(c) $c \in X_k$; or

(d) $x_i + c \in X_k$.

Our procedure then works on sentences such as the following:

$$\exists x_1 \exists x_2 (\neg(3 < x_1) \wedge \neg(4 < x_2)$$

$$\forall x_1 \forall x_2 (x_1 + 1 < x_2 \rightarrow \exists x_3 (x_1 < x_3 \wedge x_3 < x_2))$$

$$\forall X_1 \exists X_2 \forall x_3 (\neg(x_3 \in X_1) \vee (x_3 + 1 \in X_2))$$

$$\forall x_1 \exists x_2 (x_2 < x_1)$$

$$\exists X_1 \exists x_2 (x_2 \in X_1 \wedge \forall x_3 (x_3 \in X_1 \rightarrow x_3 + 1 \in X_1))$$

$$\exists X_1 \forall x_2 (x_2 \in X_1).$$

In English, these say

- Natural numbers less than five exist.
- For every pair of natural numbers more than two apart there is another in between.
- For every finite set there is a corresponding finite set shifted over one place.
- There is no smallest natural number.
- There is a finite set that satisfies the conditions of mathematical induction.
- There is a finite set containing every natural number.

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.

1.2.2 $S2S$

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 [8].

1.3 Outline

Our goals for this undertaking are numerous. First, we wish to simply restate Büchi's proof that $WS1S$ is decidable using the terminology and notation of modern automata theory. We do so in chapter 2.

Second, we wish to determine if the decision procedure is as slow in practice as the worst case would lead us to believe. This will cast light on how reasonable a modified version of Hilbert's program would be: if the practical inputs for $WS1S$, the weakest theory in the hierarchy, exhibit running times remotely near that of the worst case then a modified version of Hilbert's program has virtually no chance of success.

Third, we want to see what effect data structures and algorithms have on the running time of the decision procedure. We have found a number of places

where the choice of data structures and algorithms has a significant effect on the running time of the algorithm, most notably in the areas of minimization and determinization.

One of the major problems we must deal with in the decision procedure is that our automata grow very quickly, hence we have explored many ways to keep our automata as small as possible. In this vein, we find that the choice of how to encode n-tuples as input tapes has a profound effect on the running time of the algorithm. We also explore how effective standard minimization techniques are. Additionally, we introduce two new strategies for minimization that rely on special properties of the automata we build. One of these is a new algorithm that minimizes certain automata in linear time.

Another major problem we have is determinizing nondeterministic automata; this is the process that takes the most time in the decision procedure. For that reason we have explored many different routes in the implementation, most notably data structures for storing sets of integers. In chapter 3 we present data to show what structures work best, at least on the moderately sized inputs we can run, and in chapter 4 we present and analyze a new data structure that, although it is not the winner for this application, may have uses elsewhere.

Fourth, we wanted to write the decision procedure so it could serve as part of a suite to test automata algorithms. Now, in addition to automata arising from string processing or natural language processing, we have an abundance of automata arising from L_{S1S} formulas to test algorithms on. We have already found one algorithm that receives good reviews in the literature that performs quite poorly on our machines.

Chapter 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 [6]. We restate the proof in full in order to express it in modern notation and language.

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 finite set X with largest element m is a string of m 0's and 1's such that the nth character is 1 if and only if $n \in X$. The encoding of the empty set is the empty string. The encoding of a natural number n is n 0's followed

by a 1.

Note that the encoding of sets does not assign meaning to every tape, in particular those ending in 0's. Likewise, the encoding of natural numbers does not assign meaning to each tape, specifically those that don't have exactly one 1, and those that don't have their only 1 at the end of the tape. We do, however, wish do give these tapes meaning, and we will do so after the following definitions.

Definition 2.1 *The empty string is denoted* e*.*

Definition 2.2 *The symbol on a zero track tape will be denoted* $\perp$*.*

Definition 2.3 $\mathcal{P}_{finite}(\mathbf{N})$ *is the set of finite subsets of* $\mathbf{N}$*.*

Definition 2.4 $\Sigma_0 = \{\perp\}$*,* $\Sigma_1 = \{0, 1\}$*, and in general* $\Sigma_n = \{0, 1\}^n$*.*

These are the alphabets of our automata. We will generally write these symbols from these alphabets as column vectors.

Definition 2.5 *Let* $f(\chi_1, \ldots, \chi_n)$ *be a formula in* L_{S1S}*. Then* $Zero(f)$ *is that element of* Σ_n *such that the* i*th component is* 0 *if* χ_i *is a set variable and* 1 *is* χ_i *is a scalar variable.*

$Zero(f)$ is a symbol we can add to the end of a tape without changing its meaning as far as f is concerned. That is, for each formula f in L_{S1S} we will build a machine M such that if M accepts w then it accepts $w \cdot Zero(f)$ as well. 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 $\mathbf{N}$ then $Zero(f) = \begin{smallmatrix}1\\1\\1\end{smallmatrix}$. If f has two variables, the first a set variable and the second a natural number variable, then $Zero(f) = \begin{smallmatrix}0\\1\end{smallmatrix}$.

Definition 2.6 *Let γ, our encoding function, which for each integer $k \geq 0$ maps $(\mathbf{N} \cup \mathcal{P}_{finite}(\mathbf{N}))^k$ to Σ_k^* (that is, γ maps k-tuples of natural numbers or finite sets of natural numbers to strings over Σ_k) be defined by*

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

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

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

(4) for $(\chi_1, \ldots, \chi_n)$, $n \geq 2$ *write* $w = w_1 \ldots w_k$ *for* $\gamma(\chi_1, \ldots, \chi_{n-1})$ *and* $u_1 \ldots u_l$ *for* $\gamma(\chi_n)$. *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* $\gamma(\chi_1, \ldots, \chi_n) = v_1 \ldots v_{\max\{k,l\}}$.

Definition 2.7 $M \in DFA(f)$ *if and only if*

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

(2) $f(\chi_1, \ldots, \chi_n)$ *is true implies* M *accepts* $\gamma(\chi_1, \ldots, \chi_n)$.

For example, figure 2 shows an automaton that corresponds to $x = y$.

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 extended inverse function defined over all of $\bigcup_{k=0}^{\infty} \Sigma_k^*$. More motivation for extending the inverse function will be given later.

Definition 2.8 *Let* $u = u_1 \ldots u_l \in \{0, 1\}^*$. γ_u^{-1} *is defined as follows:*

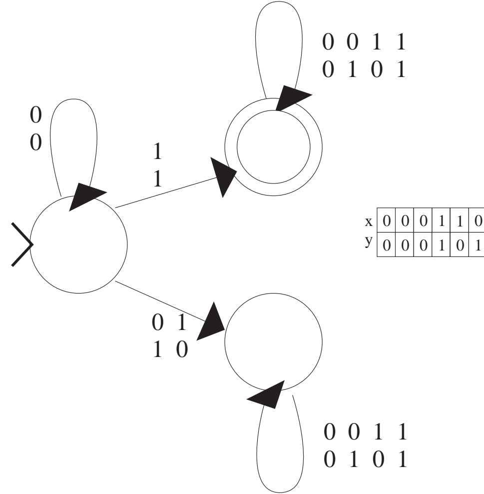


Figure 2.1: An automaton in $DFA(x = y)$ and an input it accepts.

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

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

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

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

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.

$$\begin{aligned} \gamma(\emptyset) &= e \\ \gamma(3) = \gamma(\{3\}) &= 0001 \\ \gamma(\{4,6\}) &= 0000101 \\ \gamma(3, 1, \{1, 2\}) &= \begin{matrix} 0\;0\;0\;1 \\ 0\;1\;0\;0 \\ 0\;1\;1\;0 \end{matrix} \end{aligned}$$

$$
\begin{aligned}
\gamma^{-1}(\bot\bot\bot) &= () \\
\gamma_{00}^{-1}\begin{pmatrix} 0\,0\,0\,0 \\ 0\,0\,0\,0 \end{pmatrix} &= (4, 4) \\
\gamma_{10}^{-1}\begin{pmatrix} 0\,0\,0\,0 \\ 0\,0\,0\,0 \end{pmatrix} &= (\emptyset, 4) \\
\gamma_{110}^{-1}\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} &= (\{2\}, \{0, 2, 3\}, 0)
\end{aligned}
$$

Definition 2.9 *Let* $a \in \Sigma_n$ *be* $\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}$. *Let* $b \in \Sigma_m$ *be* $\begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$. *If* $m = 0$ *then* $stack(a, b) = a$. *For* $m > 0$, $stack(a, b) = stack(\begin{pmatrix} a_1 \\ \vdots \\ a_n \\ b_1 \end{pmatrix}, \begin{pmatrix} b_2 \\ \vdots \\ b_m \end{pmatrix})$.

That is, $stack(a, b)$ is that element of Σ_{n+m} with its top n components from a and bottom m components from b. We can extend the *stack* function to strings:

Definition 2.10 *Let* $w \in \Sigma_n^*$ *be* $w_1 \cdots w_k$ *and* $u \in \Sigma_m^*$ *be* $u_1 \cdots u_k$. *Then define* $stack(w, u)$ *to be a string in* Σ_{n+m}^* *and to equal* $stack(w_1, u_1) \cdot \cdots \cdot stack(w_k, u_k)$.

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 the 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.

For an atomic formula of the form $c < x$, where c is a fixed natural number, we build the automaton in figure 2.1. Formally, let

$$M = (\{q_0, \ldots, q_{c+1}, no\}, \Sigma_1, \delta, q_0, \{q_{c+1}\})$$

where δ is defined by:

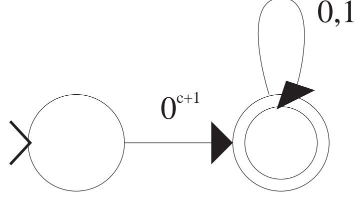


Figure 2.2: An automaton in $DFA(c < x)$.

- $\delta(q_i, 0) = q_{i+1}$ for $0 \leq i < c+1$,
- $\delta(q_i, 1) = no$ for $0 \leq i < c$,
- $\delta(q_{c+1}, 0) = \delta(q_{c+1}, 1) = q_{c+1}$,
- $\delta(no, 0) = \delta(no, 1) = no$.

Then $M \in DFA(c < x)$: suppose that $c < x$. Then $\gamma(x) = 0^x 1$ and we have

$$(q_0, 0^x 1) \vdash_M (q_1, 0^{x-1} 1) \vdash_M \cdots \vdash_M (q_{c+1}, 0^{x-c-1} 1).$$

But q_{c+1} has transitions to itself and is an accepting state, so we also have

$$(q_0, \gamma(x)) \vdash_M^* (q_{c+1}, e).$$

In other words, M accepts $\gamma(x)$. Now suppose that M accepts $w = w_1 \ldots w_n$. Then we must have

$$(q_0, w_1 \ldots w_n) \vdash_M (q_1, w_2 \ldots w_n) \vdash_M \cdots \vdash_M (q_c + 1, w_{c+2} \ldots w_n)$$

since the only path from q_0 to the accepting state is through q_1, q_2, etc. But then we must have $w_1 = w_2 = \cdots = w_{c+1} = 0$, and so $\gamma^{-1}(w)$, which is determined by the position of the first 1, is at least $c+1$ and hence $\gamma^{-1}(w) > c$ and therefore satisfies $c < x$.

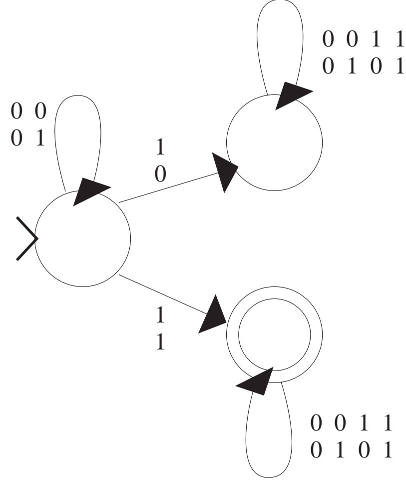


Figure 2.3: An automaton in $DFA(x_1 \in X_2)$

.

For a formula of the form $x_1 + c \in X_2$ where $c \geq 0$ we build the automaton shown in figure 2.1 if $c = 0$ and the (nondeterministic) automaton shown in figure 2.1 otherwise. Formally, in the latter case we have, after determinization, $M = (\{q_1, \ldots, q_c, yes, no\}, \Sigma_2, \delta, q_1, \{yes\}$ where δ is given by

- $\delta(q_1, {}^0_0) = \delta(q_1, {}^1_0) = q_1$,
- $\delta(q_1, {}^1_0) = \delta(q_1, {}^1_1) = q_2$,
- $\delta(q_i, \sigma) = q_{i+1}$ for all $\sigma \in \Sigma_2$ and $2 \leq i < c$,
- $\delta(q_c, {}^0_1) = \delta(q_c, sxx) = yes$,
- $\delta(q_c, {}^0_0) = \delta(q_c, sxo) = no$,
- $\delta(yes, \sigma) = yes$ for all $\sigma \in \Sigma_2$,
- $\delta(no, \sigma) = no$ for all $\sigma \in \Sigma_2$,

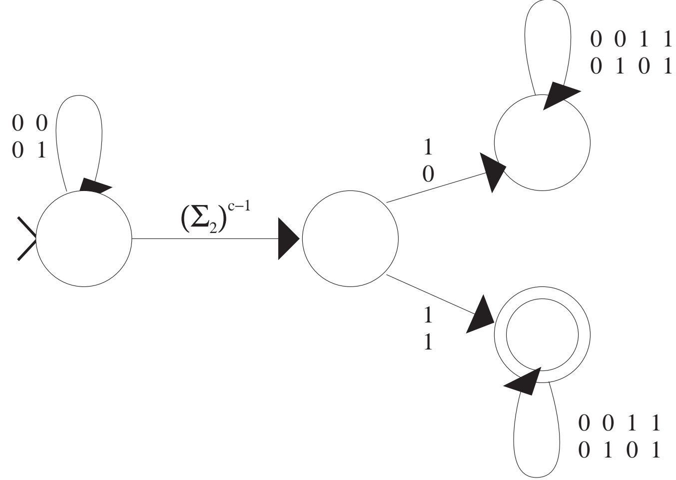


Figure 2.4: An automaton in $DFA(x_1 + c \in X_2)$.

from which we get $M \in DFA(x_1 \in X_2)$. In the former case we have $M = (\{s, yes, no\}, \Sigma_2, \delta, s, \{yes\})$ where δ is given by the following table:

q	$\substack{0\\0}$	$\substack{0\\1}$	$\substack{1\\0}$	$\substack{1\\1}$
s	s	s	no	y
yes	yes	yes	yes	yes
no	no	no	no	no

and in this case too we have $M \in DFA(x_1 < X_2)$.

For an atomic formula of the form $x_1 + c < x_2$, where again $c \geq 0$, we build the nondeterministic automaton in figure 2.1. The deterministic equivalent is $M = (\{q_0, \ldots, q_{c+1}, no\}, \Sigma_2, \delta, q_0, \{q_{c+1}\})$, where δ is defined by

- $\delta(q_0, \substack{0\\0}) = q_0$,
- $\delta(q_0, \substack{1\\0}) = q_1$,
- $\delta(q_0, \substack{0\\1}) = \delta(q_0, \substack{1\\1}) = no$,
- $\delta(q_i, \substack{0\\0}) = \delta(q_i, \substack{1\\0}) = q_i + 1$ for $1 \leq i \leq c$,

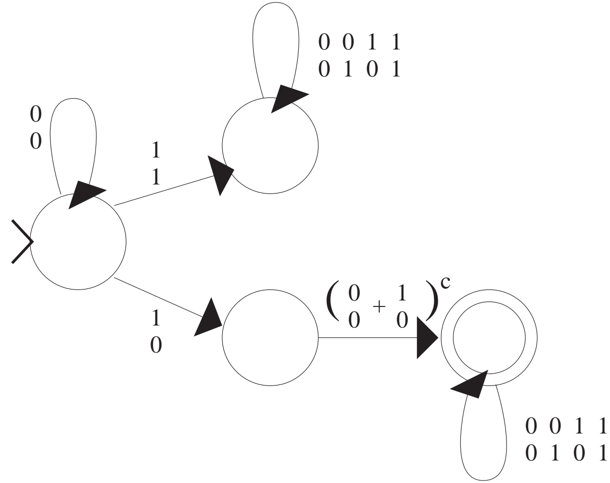


Figure 2.5: An automaton in $DFA(x_1 + c < x_2)$.

- $\delta(q_i, {0 \atop 1}) = \delta(q_i, {1 \atop 1}) = no$ for $1 \leq 1 \leq c$,
- $\delta(q_{c+1}, \sigma) = q_{c+1}$ for all $\sigma \in \Sigma_2$,
- $\delta(no, \sigma) = no$ for all $\sigma \in \Sigma_2$.

$M \in DFA(x_1 + c < x_2)$.

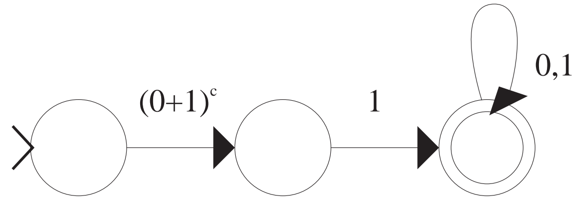


Figure 2.6: An automaton in $DFA(c \in X)$.

Finally, given an atomic formula of the form $c \in X$ where c, as usual, is at least zero, we build the deterministic equivalent of the nondeterministic automaton pictured in figure 2.1. Formally, this is $M = (\{q_0, \ldots, q_{c+1}, no\}, \Sigma_1, \delta, q_0, \{q_{c+1}\})$ where the transition relation is such that

- $\delta(q_i, 0) = \delta(q_i, 1) = q_{i+1}$ for $0 \leq i < c$,

- $\delta(q_c, 0) = no$,
- $\delta(q_c, 1) = q_{c+1}$,
- $\delta(q_{c+1}, 0) = \delta(q_{c+1}, 1) = q_{c+1}$,
- $\delta(no, 0) = \delta(no, 1) = no$.

From this definition we can see that $M \in DFA(c \in X)$.

2.2 Automata for Non-atomic Formulas

2.2.1 Logical Connectives

Consider a formula $f(\chi_1, \ldots, \chi_n)$ that is of the form $g(\chi_1, \ldots \chi_n) \wedge h(\chi_1, \ldots \chi_n)$ or $g(\chi_1, \ldots \chi_n) \vee h(\chi_1, \ldots \chi_n)$. Note that we can, without loss of generality, assume that g and h work on exactly the same variables: if they don't, we can just pad them with dummy variables (if g is $x_1 < x_3$ and h is $x_1 \in X_2$ then we would consider instead $(x_1 < x_3) \wedge (0 \in X_1 \vee 0 \notin X_1)$ and $x_1 \in X_2 \wedge (x_3 \in X_2 \vee x_3 \notin X_2)$). Constructing a machine $F \in DFA(f)$ is easy: 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 accepting either $L(G) \cap L(H)$ or $L(G) \cup L(H)$.

Theorem 2.11 *If $G \in DFA(g)$, $H \in DFA(h)$ and $L(F) = L(G) \cap L(H)$ (respectively, $L(F) = L(G) \cup L(H)$), then $F \in DFA(g \wedge h)$ (respectively, $F \in DFA(g \vee h)$).*

Proof. Suppose g and h are as given and f is the conjunction (respectively, disjunction) of g and h. Suppose $w \in L(G) \cap L(H)$ ($w \in L(G) \cup L(H)$). Then G accepts w and (or) H accepts w. Since $G \in DFA(g)$ and $H \in DFA(h)$ we then have $\gamma^{-1}(w)$ makes g true and (or) $\gamma^{-1}(w)$ makes h true, whence f is true.

Suppose $(\chi_1, \ldots, \chi_n)$ makes f true. Then $(\chi_1, \ldots, \chi_n)$ makes g and (or) h true as well. Since $G \in DFA(g)$ and $H \in DFA(h)$ we then have G and (or) H accept $\gamma(\chi_1, \ldots, \chi_n)$. But then F also accepts $\gamma(\chi_1, \ldots, \chi_n)$. □

For a formula of the form $f(\chi_1, \ldots, \chi_n) = \neg g(\chi_1, \ldots, \chi_n)$ we can construct $G \in DFA(g)$ and make its final states non-final and vice-versa to get F:

Theorem 2.12 *If $G \in DFA(g)$ and $L(F) = \bar{L}(G)$ then $F \in DFA(\neg g)$.*

Proof. Let F and G be as given. Suppose F accepts w. Then G does not accept w. Since $G \in DFA(g)$ we then have that $\gamma^{-1}(w)$ does not satisfy g. Then $\gamma^{-1}(w)$ does satisfy $\neg g$. Suppose that $(\chi_1, \ldots, \chi_n)$ satisfies $\neg g$. Then it does not satisfy g and therefore $\gamma(\chi_1, \ldots, \chi_n)$ is not accepted by G and so must be accepted by F.
□

2.2.2 Quantifiers

Finally, we consider formulas involving quantifiers. Let $g(\chi_1, \ldots, \chi_n)$ be an n-ary function and let $h(\chi_1, \ldots \chi_{n-1}) = (\exists \chi_n)(g(\chi_1, \ldots \chi_n))$ (since we can write $(\forall \chi_n)(g(\chi_1, \ldots, \chi_n))$ as $\neg(\exists \chi_n)(\neg g(\chi_1, \ldots, \chi_n))$, 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 the nondeterministic finite automaton

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

where

$$\Delta = \{(q, \sigma, q') \in K \times \Sigma_{n-1} \times K \mid \delta\,(q, stack(\sigma, 0)) = q' \vee \delta\,(q, stack(\sigma, 1)) = q'\}$$

and

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

Note that $F \subset F'$ since $w \in Zero(h)^*$ and for any $f \in F$ we have $(f, e) \vdash_H^* (f, e)$.

H is essentially the same automaton as G, except the bottom track has been erased from all the symbols on its transitions and the set of final states has been altered somewhat. The result is most likely a nondeterministic automaton.

Lemma 2.13 *Let $G = (K, \Sigma_n, \delta, s, F) \in DFA(g)$ and let H be constructed as above. Then*

$$(\exists u \in \Sigma_1^*)[(q, stack(w, u)) \vdash_G^* (q', e)]$$

if and only if $(q, w) \vdash_H^* (q', e)$.

Proof. Suppose the former. Write w as $w_1 \ldots w_k$ and u as $u_1 \ldots u_k$. Then there must be a sequence of states $q_0, q_1, \ldots, q_k$ with $q_0 = q$ and $q_k = q'$ such that whenever $1 \leq i \leq k$ we have

$$(q_{i-1}, stack(w_i \ldots w_k, u_1 \ldots u_k)) \vdash_G (q_i, stack(w_{i+1} \ldots w_k, u_{i+1} \ldots u_k)),$$

which means that

$$\delta\left(q_{i-1}, stack(w_i, u_i)\right) = q_i.$$

But then by the construction of Δ we have

$$(q_{i-1}, w_i, q_i) \in \Delta$$

from which we obtain

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

and finally,

$$(q_0, w_1 \ldots w_k) \vdash_H^* (q_k, e)$$

Since $q_0 = q$, $q_k = q'$ and $w_1 \ldots w_k = w$ we have $(q, w) \vdash_H^* (q', e)$.

Now suppose that $(q, w) \vdash_H^* (q', e)$. Again, write w as $w_1 \dots w_k$. By the definition of the $\vdash_H^*$ relation, there must exist states $q_0, \dots, q_k \in K$ with $q_0 = q$ and $q_k = q'$ such that, for all i satisfying $1 \leq i \leq k$,

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

which means

$$(q_{i-1}, w_i, q_i) \in Delta.$$

This means, because of the way we constructed Δ, that for each i, $1 \leq i \leq k$, there exists a $u_i \in Sigma_1$ such that

$$\delta\left(q_{i-1}, stack(w_i, u_i)\right) = q_i$$

and so

$$(q_{i-1}, stack(w_i \dots w_k, u_i \dots u_k)) \vdash_G (q_i, stack(w_{i+1} \dots w_k, u_{i+1} \dots u_k)).$$

Chaining these together gets us

$$(q_0, stack(w_1 \dots w_k, u_1 \dots u_k)) \vdash_G^* (q_k, e).$$

Since $q_0 = q$, $q_k = q'$, and $w_1 \dots w_k = w$, we have found a $u \in \Sigma_1^*$ (namely $u_1 \dots u_k$) such that $(q, (stackw, u)) \vdash_G^* (q', e)$. □

Claim 2.14 *$H \in DFA(h)$.*

Proof. First, suppose $h(\chi_1, \dots, \chi_{n-1})$ is true. Then there exists a χ_n such that $g(\chi_1, \dots, \chi_n)$ is true. $G \in DFA(g)$ so G must accept $\gamma(\chi_1, \dots, \chi_n)$. Note that we can write that encoded vector as $stack(w, u)$, where $w \in \Sigma_{n-1}$ encodes χ_1 through χ_{n-1} and $u \in \Sigma_1^*$ encodes χ_n.

Since G accepts $stack(w,u)$ we have $(s, stack(w,u)) \vdash_G^* (f,e)$ where $f \in F$. By lemma 2.13 we then have $(s,w) \vdash_H^* (f,e)$. Since $f \in F$ and $F \subseteq F'$ we know that H accepts w. What we want is that H accepts $\gamma(\chi_1, \ldots, \chi_{n-1})$, so we need to figure out how that and w relate. Recall that $stack(w,u)$ was the encoding of $(\chi_1, \ldots, \chi_n)$. Be the definition of γ, that encoding is obtained from the encodings of $(\chi_1, \ldots, \chi_{n-1})$ and χ_n by padding one of them if necessary. If the latter was padded then $w = \gamma(\chi_1, \ldots, \chi_{n-1})$ and we are done. If the former was padded then we have that $\gamma(\chi_1, \ldots, \chi_{n-1})$ is a prefix of w, say $w_1 \ldots w_l$. In that case $w_{l+1} = \cdots = w_k = Zero(h)$ and since

$$(q_l, w_{l+1} \ldots w_k) \vdash_H^* (q_k, e)$$

we get $q_l \in F'$ by the construction of F' and hence H accepts $w_1 \ldots w_l$ which is the encoding of $(\chi_1, \ldots, \chi_{n-1})$.

Now suppose that H accepts w. We then have $(s,w) \vdash_H^* (f',e)$ for some $f' \in F'$. By the construction of F' there must be a $z \in Zero(h)^*$ such that $(f',z) \vdash_H^* (f,e)$ for some $f \in F$. By the properties $\vdash_H^*$ we then have $(s,wz) \vdash_H^* (f,e)$. Now, by lemma 2.13 (this time in the other direction), we can find a $u \in \Sigma_1^*$ such that $(s, stack(w \cdot z, u)) \vdash_G^* (f,e)$. and therefore G accepts $stack(w \cdot z,)$. Since $G \in DFA(g)$ we then have $\gamma^{-1}(stack(w \cdot z, u))$ satisfies g. But $\gamma^{-1}(stack(w \cdot z, u)) = (\gamma^{-1}(w \cdot z), \gamma^{-1}(u))$ and $\gamma^{-1}(w \cdot z) = \gamma^{-1}(w)$ since $z \in Zero(h)^*$ and we chose $Zero(h)$ so it would not change the meaning of a tape.

We have established that $g(\gamma^{-1}(w), \gamma^{-1}(u \cdot v))$ is true. Therefore there exists a χ_n, namely $\gamma^{-1}(u \cdot v)$ that makes $g(\gamma^{-1}(w), \chi_n)$ true, hence $\gamma^{-1}(w)$ satisfies h. $\square$

Note that it is necessary to modify the set of final states. This is best illustrated by an example (Figure 2.7). The last automaton rejects e when it should accept

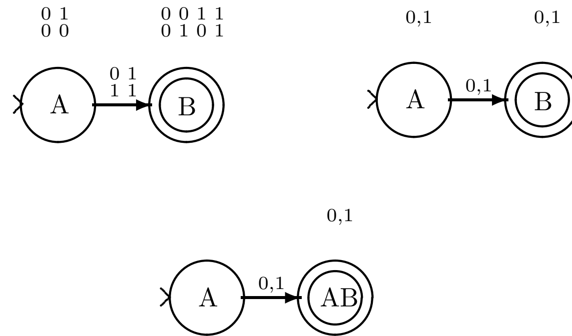


Figure 2.7: Result of erasing a track without altering final states

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.

2.3 Testing membership in $WS1S$

Now for any sentence f 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).

Chapter 3

Implementation

In the previous chapter we presented a proof that $WS1S$ is decidable. Being mathematical, the proof omits details of how the structures mentioned therein are represented inside a computer, and how the procedures used are performed algorithmically. This chapter deals with minor issues from a few general areas. Chapters 4 and 5 are devoted to the very important issues of determinizing and minimizing our automata.

3.1 The Encoding Function and Automata for Atomic Formulas

Whatever encoding function is used will determine what automata are built for atomic formulas. We stated before that we needed to extend the inverse of the encoding function so that it gave meaning to every tape. Here we discuss the reason for that and several options for how to do the encoding.

When we construct the nondeterministic machine H what we want is a machine that accepts $\gamma(\chi_1, \ldots, \chi_{n-1})$ if and only if there is a χ_n such that we can add $\gamma(\chi_n)$

as the bottom track of $\gamma(\chi_1, \ldots, \chi_{n-1})$ and get a string accepted by G. But what we are really doing is constructing H so that it accepts $\gamma(\chi_1, \ldots, \chi_n)$ if and only if there is some arbitrary string in Σ_1^* that we can add to get a new string accepted by G. Remember that we have shown that γ is not onto, so it is possible that the string we could add doesn't correspond to any actual number. We would, in effect, be proving our existentially quantified formula true by saying that there is a χ_n, namely garbage that satisfies the quantified predicate.

To avoid such situations, we need to assign a meaning to those tapes that are not in the range of γ. If a track is to be interpreted as a natural number, then it would have no meaning if it had no 1 on it or if it had characters after the first 1. Tracks that represent sets have would have no inverse under γ if they had trailing zeros. We have considered three solutions to this problem.

The first solution is to change the formulas instead of extending the inverse function. Consider a machine M_1 that works on an n-track tape. We want to construct a machine M_2 that accepts $(n-1)$-track tapes only if to them we can attach a meaningful bottom track to get a tape accepted by M_1. That would correspond to the function $f_2(\chi_1, \ldots, \chi_{n-1}) = \exists(\chi_n)[f_1(\chi_1, \ldots, \chi_n) \wedge meaningful(\chi_n)]$. We would then have to construct a machine for $meaningful(\chi_n)$, which would not be hard to do. However, after converting all n existential quantifiers this way we would end up with $\bigwedge_{i=1}^{n} meaningful(\chi_i)$, which would require $O(2^n)$ states. Such large automata are to be avoided whenever possible, so we abandoned this strategy early on.

The next two solutions involve extending the inverse function. Both extensions ignore trailing zeros for tracks encoding sets and ignore everything after the first 1 for tracks encoding natural numbers. They differ in how they interpret such tracks

that contain no 1's.

The encoding we presented in chapter 2 interprets that track according to its length, essentially pretending that there is a 1 just off the end of the tape. This leads to compact machines: the automaton for

$$\bigvee_{i=1}^{n} x_i \neq 0 \tag{3.1}$$

has only three states.

An alternate encoding would interpret the same track as 0. Using this encoding, we would get a machine with 2^n states for formula 3.1. We would also get a different set of automata for atomic formulas including the one shown in figure 3.1. The

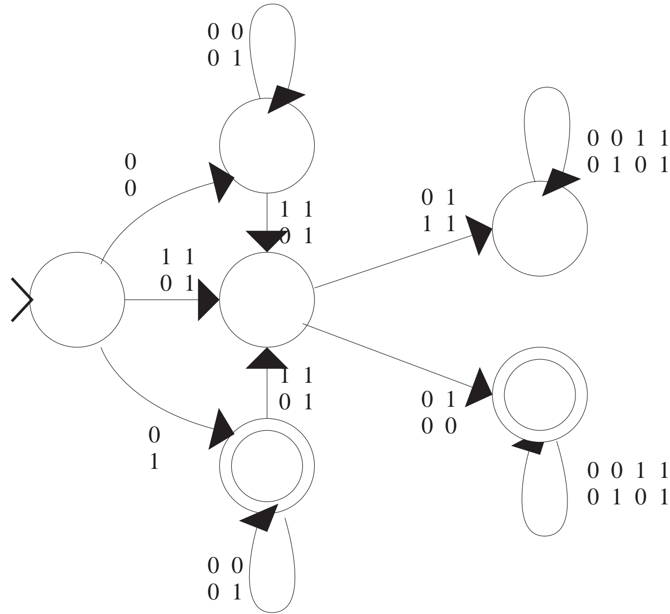


Figure 3.1: Another possible machine for $x_1 \in X_2$.

difference between this one and the one in figure 2.1 is not exponential, but clearly this one is more complicated.

Clearly the size of the basic machines is going to have a profound effect on the running time of the decision procedure. To illustrate this, we have run the

decision procedure using both encodings on one group of 100 "easy" and 20 "hard" randomly generated formulas. (Both groups were generated by the same process. Here "easy" simply means "takes under 5 seconds to decide.") The running time using the first encoding is 12.4 seconds on the "easy" inputs and 139 seconds on the "hard" ones, while for the second encoding we get 47.0 seconds and 975 seconds, obviously drastically worse. It also seems that the harder the formulas get, the more important it is to use a good encoding.

3.2 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. When we combine machines for these subformulas, we get large machines, some sections of which may not check certain tracks of the tape at all. We formalize this notion with the definition below.

Definition 3.1 *A state q in an n-track automaton $M = (K, \Sigma_n, \delta, q, F)$ is said to be insensitive to track i if, for all $\sigma_1 \in \Sigma_{i-1}$ and all $\sigma_2 \in \Sigma_{n-i-1}$ we have $\delta(q, stack(\sigma_1, 0, \sigma_2)) = \delta(q, stack(\sigma_1, 1, \sigma_2))$.*

So what we have said above is that we often encounter states that are insensitive to many tracks. For example, if x_i is a scalar variable and $(s, w \cdot v) \vdash_M^* (q, v)$ for some string w that contains a 1 somewhere on track i, then q must be insensitive to track i since our encoding ignores anything after the first 1 on track i. Further, any state reachable from q must be insensitive to track i for the same reason.

We store the complete transition function in an array for each state, which would normally require 2^n entries. Retrieving $\delta(q, \sigma)$ from those 2^n entries is

simple: all we must do is convert σ into the index of an array by treating it as the binary representation of the index. But if state q is insensitive to m tracks, then the transition function at q requires only 2^{n-m} units of storage. This is desirable to cut down on the total memory required to represent our machines (so we can process more complex formulas in the same amount of space) and because it will speed up the decision procedure in some places (for example, if we need to examine all entries in the transition table). We will, however, need a more complex process to retrieve $\delta(q, \sigma)$ from this compressed space.

With our space improvement, we must mask out certain components of σ before converting to an array index. What we need then is a function $f_q : \{0,1\}^n \to \{0, \ldots, 2^{n-m} - 1\}$ that satisfies

$$f_q(b_1, \ldots, b_{i-1}, 0, b_{i+1}, \ldots, b_n) = f_q(b_1, \ldots, b_{i-1}, 1, b_{i+1}, \ldots, b_n)$$

for any i such that q is insensitive to track i. Since the transition function is evaluated so often, it is critical that this function is computed as quickly as possible; otherwise the performance of the decision procedure will be degraded greatly.

Note that elementary bitwise operations will not do the trick, since their range is $\{0, \ldots, 2^n - 1\}$. What we can do, however, is simply delete any components of σ that q is insensitive to. For example if σ is $\mathit{stack}(1,0,1,1,0,1,1,0)$ and we are interested in only the second, fourth, fifth, and eighth bits (with the most significant bit numbered one), we mask out the ignored 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. No commonly used CPU has an operation that will perform this in one instruction, so we will have to write algorithms to compute f_q. One 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 last 8 bits in 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 to work on next 8 bits
  symbol := symbol / 2^8      // do the same for symbol
end for
```

Again, we have run the decision procedure using both methods on groups of hard and easy inputs. For easy inputs, we have a total time of 13.0 seconds for the first algorithm and 10.1 for the second. On hard inputs we get 211 seconds using the first and 171 using the second.

3.3 Algorithm for Union and Intersection

Suppose f is of the form $g \wedge h$ or $g \vee h$ and that we have machines $G \in DFA(g)$ and $H \in DFA(h)$. The machine F that we build corresponding to f needs to accept $L(G) \cap L(H)$ or $L(G) \cup L(H)$. We can use the standard cross-product formulation of F to get what we need, but many states in the resulting automaton will be unreachable. Naturally we want to avoid generating unreachable states. We can do so by building F forward from its start state (s_G, s_H) (where s_G and s_H are the start states of G and H respectively) and finding the states that are reachable from it. In the next step we find those states that are reachable in two steps from (s_G, s_H), and so forth. This method requires us to use a structure to keep track of what states we have seen so far; we use an xy-tree for this purpose.

Chapter 4

Determinization

It is the determinization algorithm that leads to the superexponential running time of the decision procedure for $WS1S$, and profiling confirms that it is that algorithm that consumes most of the procedure's time. Naturally, we have put much effort into finding ways to improve the running time (although, of course, we can do nothing about the exponential nature of the worst case). To that end, we have studied numerous data structures that the algorithm can employ as well as two minor variations in the algorithm, one dealing with how the set of final states in the new machine is computed and the other with how multiple quantifiers are handled.

4.1 Final States of Our Nondeterministic Automata

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 after anything else is done, as was implied in our definition of F', or we can modify the final states of the original automaton and then introduce the nondeterminism. That corresponds to letting the new set of final states be

$$F'' = \{q \in K \mid (\exists w \in Zero(h)^*, u \in \Sigma_1^*, f \in F)\,[(q, stack(w,u)) \vdash_G^* (f,e)]\}$$

instead of the original

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

We can do things either way because $F' = F''$; the justification follows directly from lemma 2.13: suppose $q \in F'$. Then Then there exist $w \in Zero(h)^*$ and $f \in F$ such that $(q,w) \vdash_H^* (f,e)$. Be lemma 2.13 there must exist a $u \in \Sigma_1^*$ such that $(q, stack(w,u)) \vdash_G^* (f,e)$ and hence $q \in F''$. Suppose $q \in F''$. Then for some $w \in Zero(h)^*$, $u \in \Sigma_1^*$, and $f \in F$ we have $(q, stack(w,u)) \vdash_G^* (f,e)$. By lemma 2.13 again, we have $(q,w) \vdash_H^* (f,e)$ and so $q \in F'$.

What states are final and what states are nonfinal has no effect on how fast the determinization algorithm is run. But suppose we have a minimized machine which we are about to make nondeterministic by erasing a track from the symbols on its transitions. If we expand the set of final states we may introduce equivalent states. If we take the time to minimize the result, we will have a smaller machine to determinize and hence a smaller possible exponential blowup. On the other hand, we will have to take to time to do the minimization.

Our trials have shown that it is somewhat faster (the improvement is under ten percent) to modify the set of final states before introducing the nondeterminism: running the decision procedure on 200 hard inputs takes 64.3 minutes when we compute the final states according to the definition of F' and 55.3 minutes when we compute the final states according to the definition of F''.

4.2 Multiple Quantifiers

Suppose we have a formula that takes the form $(\exists\chi_1, \ldots, \chi_m)[g(\chi_1, \ldots \chi_n)]$. We have two choices as to how to handle these m consecutive quantifiers: we could handle them in one fell swoop by erasing from the symbols on the transitions all m tracks at once, or we could erase the tracks one by one.

If we erase all m tracks at once, we must check 2^m values of the transition function of the old machine when computing one value of the transition function at a state in the new machine. If we erase only one track at a time, we have only two old transitions to check in order to compute a new one, but we will be building several machines. In addition, it is possible that the intermediate machines are very large.

Suppose that before dealing with any quantifiers we have a k state machine. If we handle all m quantifiers at once we know we will end up with at most 2^k states. If we handle quantifiers one at a time, we could end up with a machine with 2^k states after the first step, 2^{2^k} states after the second step, and so on. Clearly this would not be a good situation. Note, however, that after eliminating i quantifiers one by one we should have a machine that is equivalent to the one we would have gotten by eliminating i quantifiers all at once, which would have at most 2^k states. Therefore, if we apply a minimization algorithm after eliminating each quantifer we will get back down to 2^k states. Still, in the worst case, we will start each step with 2^k states, get one with up to 2^{2^k} states and minimize back to 2^k. Clearly that would not be a good situation.

To determine which strategy is better, we turn back to our group of 200 hard inputs. When handling consecutive quantifiers all at once the total time for the determinization algorithm is 54.5 minutes. When doing consecutive quantifiers

one at a time the total time is 52.2 minutes, but those results do not include the times for three inputs for which the all-at-once method worked but the step-by-step method ran out of memory after about five minutes of processing. There was nothing particularly difficult about the three failed inputs; each was handled in about 15 seconds by the all-at-once method. This seems to confirm our fears that the intermediate machines in the step-by-step method can get unruly. We have found no pattern to which method works best on which inputs, so we stick with the all-at-once method to avoid using up our system resources.

4.3 Maintaining Reachable States

Each nondeterministic machine that is created must be determinized. As in the intersection and union algorithms, we compute only those state sets that are reachable from the start state of the nondeterministic machine, hence one of our major concerns is what Wood and Johnson call reachable-state-set maintenance [5]. 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.

4.3.1 Finding a state set

We have implemented three solutions to the former subproblem. State sets can be stored in either a standard binary search tree, a binary tRIe, or in an n-ary tRIe. These structures are illustrated in figures 4.3.1, 4.3.1, and 4.3.1, in which we have shown how the sets $\{2,3\},\{2,4\},\emptyset,\{3\},\{1,2,4\},\{2,3,4\}$, and $\{1,2,3\}$ would be stored if they were inserted in that order.

To store state sets in a binary search tree, we order sets using the standard

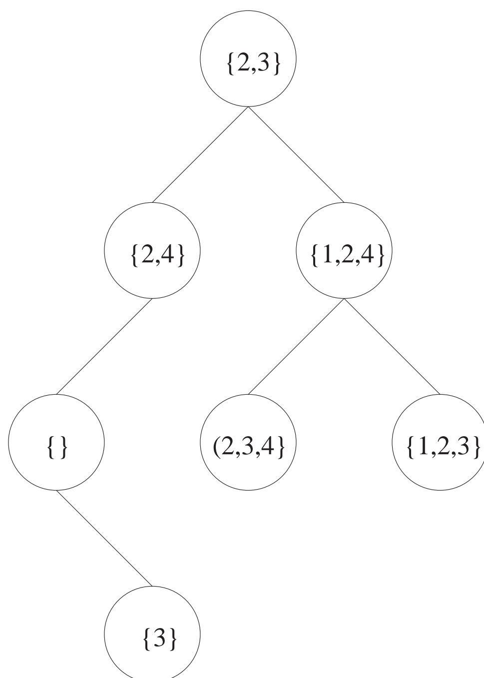


Figure 4.1: Seven sets stored in a binary search tree.

lexicographical ordering of their characteristic functions written out as strings of n 0's and 1's, where n is the maximum size of the sets. The problem with the binary search tree structure is that to find a particular set we may have to compare elements several times. For eacmple, in figure 4.3.1, to find the set $\{1, 2, 3\}$ we would first have to compare $\{1, 2, 3\}$ with $\{2, 3\}$, finding that the latter is lexicographically before the former because it does not contain 1. We then compare $\{1, 2, 3\}$ with $\{1, 2, 4\}$ and again have to test if 1 is in the set. Getting a matching answer this time, we proceed to where we find $\{1, 2, 3\}$.

To store the state sets in a tRIe, we again associate with each set the string of n 0's and 1's we get from its characteristic function. This time, however, all the sets are stored in leaves. With each internal node we associate a potential element of the sets. The internal nodes are arranged such that under their right branches all

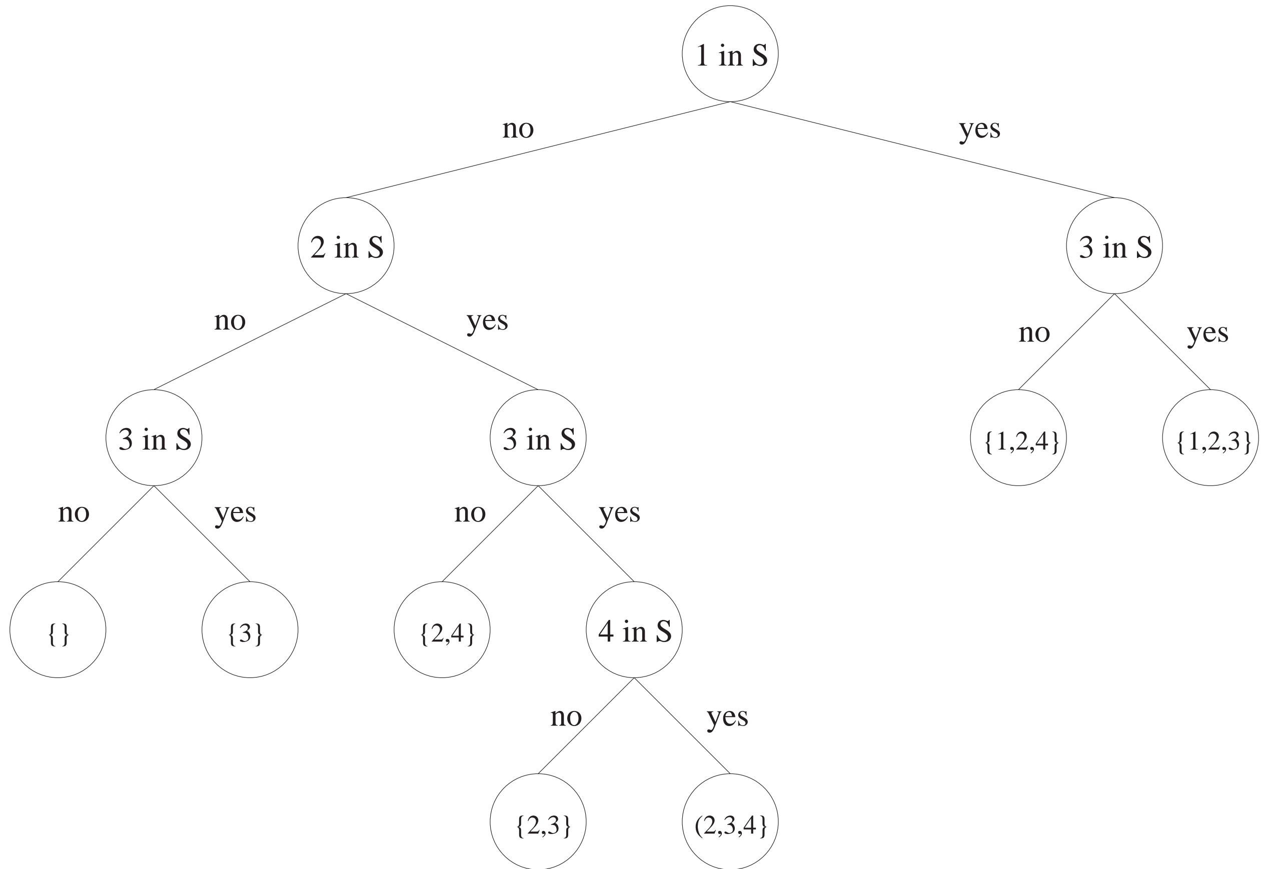


Figure 4.2: Seven sets stored in a binary tRIe.

the leaves will contain that element and under their left branches none of the leaves will. The tRIe structure is an improvement over the binary search tree because to find a particular set we will not have to consider the same potential element more than once.

For the n-ary tRIe we again associate each set with a string, but instead of a string of n 0's and 1's we use a variable length string using characters from $\{0, \ldots, n-1\}$ that is obtained by writing the elements of each set in sorted order. Again, the sets are stored only in leaves, but now the structure of the internal nodes is such that with each we associate a question of the form "what is the ith

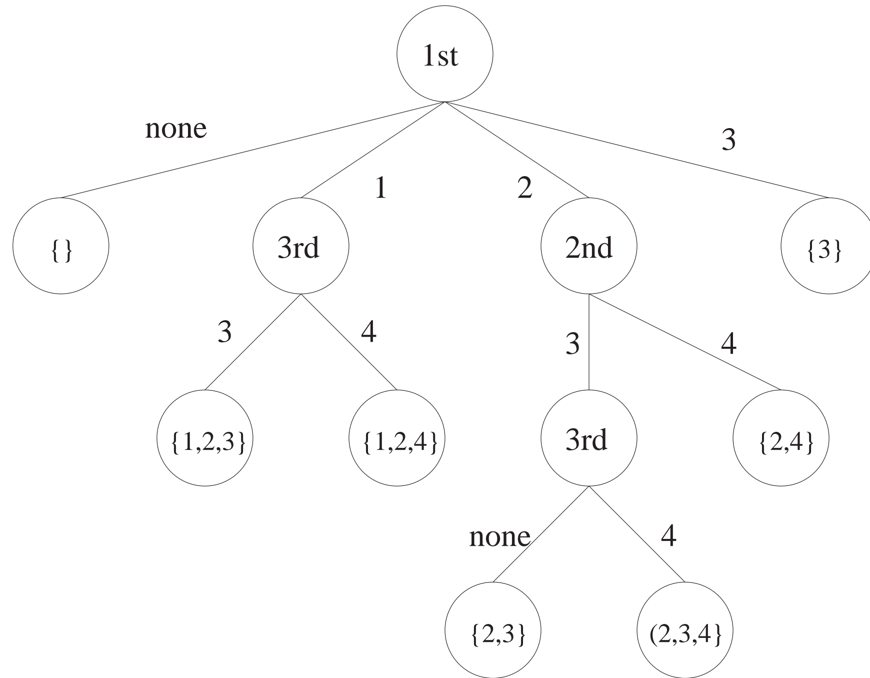


Figure 4.3: Seven sets stored in an n-ary tRIe.

character in the string" (or, equivalently "what is the ith element in the set"). There are $n - i + 2$ possible answers: there is no ith element, it is $i - 1$, it is i, ..., it is $n - 1$. Under each branch of the internal node will be sets with the same ith element. We can navigate through this sort of structure by repeatedly asking "what is the next element in the set." Note that, for any of our structures, once we have gotten to a leaf we still must confirm that we have found what we are looking for. To do that we must compare sets, and there is no better way to compare sets than by sorting them. In other words, to navigate through the n-ary tRIe, we are only doing work we would have had to do anyway. Data presented after the next section will support the claim that this last structure is our best performer.

4.3.2 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
bit vector	$O(1)$	$O(1)$	$O(n)$	n
linked list	$O(1)$	$O(m)$	$O(m)$	$64m$
sorted linked list	$O(m)$	$O(m)$	$O(m)$	$64m$
binary tree	$O(\log m)$	$O(\log m)$	$O(m)$	$160m$
hybrid	$O(\log n)$	$O(1)$	$O(m \log n)$	$2n$ to $4n$
red-black tree	$O(\log m)$	$O(\log m)$	$O(m)$	$193m$
sorted array	$O(\log m)$	$O(\log m)$	$O(m)$	$32m$

Space is measured in bits. 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 marked "hybrid" refers to a cross between a bit vector and a tree that is explored further in section 4.4.

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.

4.3.3 Results

We ran 200 hard and 13000 inputs through our decision procedure for each combination of structures for storing state sets and sets of state sets. The first table shows total time for the determinization algorithm in minutes for the hard inputs. The second table shows time in seconds for the easy inputs.

HARD	BST	tRIe	n-ary tRIe
Binary Tree	85.5	70.4	62.0
Bit Vector	110	56.3	55.3
8-ary HBV	122	76.0	63.5
Red-Black Tree	88.7	72.7	64.2
Sorted Array	77.6	65.3	56.3

EASY	BST	tRIe	n-ary tRIe
Binary Tree	720	678	634
Bit Vector	769	625	588
8-ary HBV	859	730	668
Red-Black Tree	726	682	639
Sorted Array	678	642	595

Note that, for each possible state set structure, the n-ary tRIe gives the best performance.

We have also divided our inputs into groups in which the number of variables, maximum constant, and number of clauses are the same. We then analyzed the results from each group in order to see how changing one of those parameters affects the running time of the decision procedure.

quantifiers	2		3		4		5	6
clauses	8	16	8	16	8	16	16	16
$max = 1$ 1	2	1	3	2	5	19	69	
$max = 2$ 1	2	2	4	3	13	43	195	
$max = 3$ 1	2	2	6	3	59	70	358	
$max = 4$ 1	3	3	10	10	50	132	777	
$max = 5$ 2	3	5	13	13	72	290	1205	
$max = 6$ 2	4	6	18	29	194	373		
$max = 7$ 2	4	10	23	19				

Times here are in hundredths of seconds and represent the 90th percentile – 10 percent of the inputs in each group took longer to run that the indicated figure.

Here we can see that by far the most important measure of the complexity of a formula is the number of quantifiers it has. Adding a quantifier increases the running time by a factor of about four. Increasing the maximum constant has a somewhat lesser effect, perhaps doubling the running time each time the constant is increased by one. The importance of the maximum constant increases as the number of quantifiers grows.

4.4 Data Structures

It is evident from the results in the previous chapter that the data structure used to represent sets of states during the determinization process has a profound effect on the speed of the decision procedure for WS1S. To determinize a DFA with n states requires a data structure to store subsets of $\{0, \ldots, n-1\}$. On top of the structure to store individual sets, we use another structure to keep track of what sets we have produced (a sort of index of sets). Just as we have different data structures available to store sets, we have different structures available to store the index of sets. These second structures need to be able to determine if two sets are equal and they need to be able to determine if a given integer is in a set (these, along with insertion, are the functions the set ADT must provide).

The size of the subsets depends on the DFA being determinized. Some DFA's yield small subsets; others yield quite large ones. Different data structures do well for different size sets. For example, the tree structures work well for small sets but not well for large sets because the time for each insertion is $O(\log m)$ where m is the size of the set. The bit vector structure works well for large sets but poorly for small sets because for large sets the $O(n)$ initialization time is offset by the $O(1)$ insertion time. We now describe a structure that we use for certain in-between cases.

4.4.1 Hierarchical Bit Vectors

The new structure is essentially a bit vector with a tree structure imposed on top of it. At the lowest level of the tree is an n-bit bit vector. Each bit at this level records the membership of a particular integer in the set. The next level up is a bit vector containing $n/2$ bits. The first of these bits records whether or not either

0 or 1 is in the set, the next bit records whether 2 or 3 is in the set, and so on. The next level in the tree consists of another bit vector, this one containing $n/4$ bits, each of which records whether or not one of four integers is in the set. The levels proceed in this manner up to the top level, which is a single bit that records whether or not the set is empty.

To view this structure as a tree, imagine a complete binary tree with n leaves at the lowest level. Each of those leaves corresponds to a number from 0 to $n-1$ and records whether or not that integer is in the set (leaves at the next level up are not used and will be filled with 0's). An internal node in the tree will contain a 1 if and only if 1 of its two children contains a 1.

Internally, this structure can be stored in a bit vector of size no more than $4n$ (about $2n$ if n is a power of two; otherwise there are wasted bits). The root of the tree is stored in bit zero. The left child of an internal node stored in bit i is stored in bit $2i+1$; its right child is stored in bit $2i+2$. Any node that is not an ancestor of a leaf at the lowest level of the tree is considered unuseful.

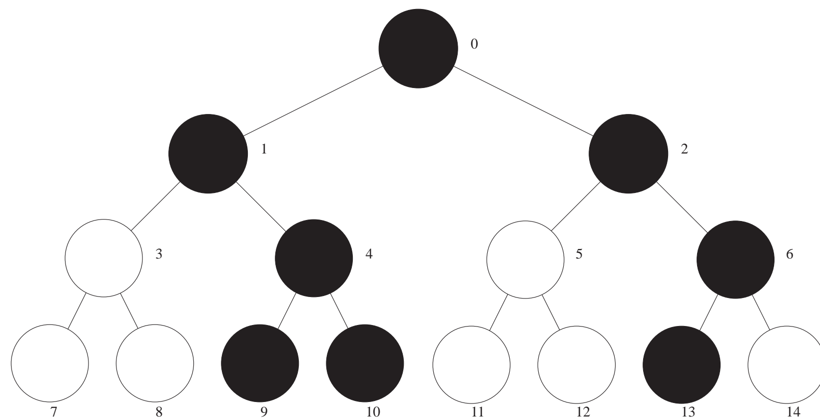


Figure 4.4: An HBV representing $\{2, 3, 6\}$

This new structure, which we will call a hierarchy of bit vectors (HBV) can

answer set membership queries in $O(1)$ time by simply examining the appropriate bit at the bottom of the tree.

To perform an insertion in an HBV, we first find the appropriate bit at the bottom level of the tree. If the bit is not set, set it and move to the parent. We continue upwards in the tree setting bits until we find a bit that is already set or we come to the root of the tree. Using this algorithm, we can perform an insertion in worst-case $O(\log n)$ time. The total time over all insertions is proportional to the number of bits which must be set, which is certainly $O(n)$.

To determine if two HBV's represent the same set we can sort the set. Our goal is to sort the set faster than can be done with a plain bit vector. The bit vector's weakness is that it has to examine each bit in turn. For small sets, it will be examining long stretches of 0's. We can overcome this by using the additional data that is stored in the HBV. Once we have found one integer in the set, we can use the information in the next-to-last level of the HBV to answer the question "is either of the next two integers in the set?". If the answer is no, we use the next level up to answer the question "is either of the next four integers in the set?". Continuing in this way, we can leap over long expanses of emptyness in a few bounds. Once the answer to one of our questions is "yes", we can go down the tree to find which integer is next.

The following procedure will determine the next integer after n that is in the set. We use various macros to help us navigate through the tree. The macro `next(i)` determines which bit to examine next if a 0 is found. For left children, this is the sibling; for right children it is the node to the right of the parent. The other macros should be self-explanatory.

```
procedure next(integer n) returns integer
```

```
  t = next(nth leaf)
  while bit t is in the useful part of the tree and is not set
    t = next(n)
  if bit t is not in the useful part of the tree
    return -1
  else
    while bit t is not a leaf
      t = leftchild(t)
      if bit t is not set
      t = sibling(t)
    end while
    find i such that t is the ith leaf at the bottom of the tree
    return i
  end if
end procedure
```

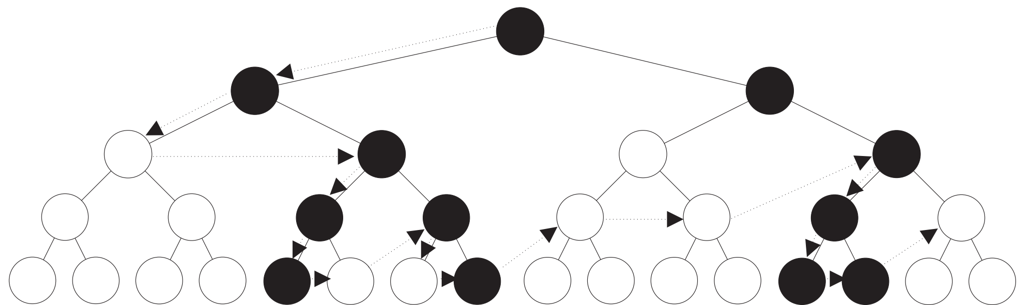


Figure 4.5: The path taken when sorting a set stored in an HBV.

We can better quantify the time each operation takes if we can get a better fix on the the size of the set in question. Let the probability that an integer is included in the set be p and let $q = 1 - p$ (then the size of the set is expected to

be pn). To make the analysis simpler, assume that $n = 2^h$.

The time taken for all insertions is equal to the number of bits that are set in the tree plus the number of times the insertion function is called. Each of the 2^h leaves of the tree is set with probability $1-q$. Each of the 2^{h-1} nodes at the second level from the bottom is set with probability $1 - q^2$. Proceeding in this manner, we can compute $I(h)$, the expected number of bits that are set in a tree of height h:

$$I(h) = \sum_{i=0}^{h} 2^{h-i}(1 - q^{2^i}).$$

If all the insertions are distinct, then there are $(1 - q)2^h$ of them on average. We add that quantity to $I(h)$ and get the table of values below.

2^h / p	0.0020	0.0078	0.0312	0.1250	0.5000
2	0.01171	0.04681	0.18652	0.73438	2.75000
4	0.03121	0.12451	0.49231	1.88257	6.43750
8	0.07796	0.30985	1.20892	4.42153	13.8711
16	0.18671	0.73763	2.81612	9.72499	28.7422
32	0.43407	1.69721	6.27019	20.4360	58.4843
64	0.98576	3.78909	13.4093	41.8719	117.969
128	2.19288	8.21174	27.8014	84.7438	236.937
256	4.77953	17.2892	56.6026	170.488	474.875
512	10.1915	35.5604	114.205	341.975	950.750
1024	21.2480	72.1204	229.410	684.950	1902.50
2048	43.4778	145.241	459.820	1370.90	3806.00
4096	87.9552	291.482	920.641	2742.80	7613.00
8192	176.910	583.963	1842.28	5486.60	15227.0
16384	354.821	1168.93	3685.56	10974.2	30455.0
32768	710.642	2338.85	7372.13	21949.4	60911.0
65536	1422.28	4678.71	14745.3	43899.8	121823.
131072	2845.57	9358.41	29491.5	87800.6	243647.
262144	5692.13	18717.8	58984.0	175602.	487295.
524288	11385.3	37436.7	117969.	351205.	974590.

Figuring out how many bit accesses are done to sort is more complex. We denote by $E_0(h)$ the expected number of bit accesses done to sort an HBV of height h when starting from the root and by $E_1(h)$ the expected number of bit accesses done to sort (again, starting from the root) given that the set is non-empty.

Let p_1 be the probability that the set is empty. Since

$$E_0(h) = p_1 \cdot 1 + (1 - p_1) \cdot E_1(h)$$

and $p_1 = q^{2^h}$ we have

$$E_1(h) = \frac{E_0(h) - q^{2^h}}{1 - q^{2^h}}.$$

Clearly $E_0(0) = E_1(0) = 1$. We now develop a recurrence for $E_0(h)$ when $h > 0$.

Let $h > 0$. Consider an HBV of height h. It consists of a root and two sub-HBV's each of height $h-1$. When sorting the HBV, we must always visit the root. With probability $1 - q^{2^{h-1}}$ the left sub-HBV is non-empty and we visit $E_1(h-1)$ nodes in it followed by $E_0(h-1)$ visits in the right sub-HBV. With probability $q^{2^{h-1}}(1 - q^{2^{h-1}})$ the left sub-HBV is empty and the right sub-HBV isn't. In this case we visit the root of the left sub-HBV and $E_1(h-1)$ nodes in the right sub-HBV. In total, we visit

$$\begin{aligned} E_0(h) &= 1 + (1 - q^{2^{h-1}})[E_1(h-1) + E_0(h-1)] \\ &+ q^{2^{h-1}}(1 - q^{2^{h-1}})[1 + E_1(h-1)] + d(h) \\ &= 1 + 2E_0(h-1) - 2q^{2^h} + d(h) \end{aligned} \tag{4.1}$$

nodes, where d is a correction needed because when the left sub-HBV is non-empty, we don't necessarily enter the right sub-HBV from its root. To complete the recurrence, we need to find an expression for d.

We will enter the left sub-HBV at one of the ancestors of its leftmost leaves (that is, somewhere along its left side). Suppose we enter the left sub-HBV at the lowest possible node (the parent of the leftmost leaves). Call that node A_1. Imagine the path that is taken through the right sub-HBV when starting from that node. Imagine also the path that would have been taken had we entered the right sub-HBV from the top. Those two paths may meet at some node and from

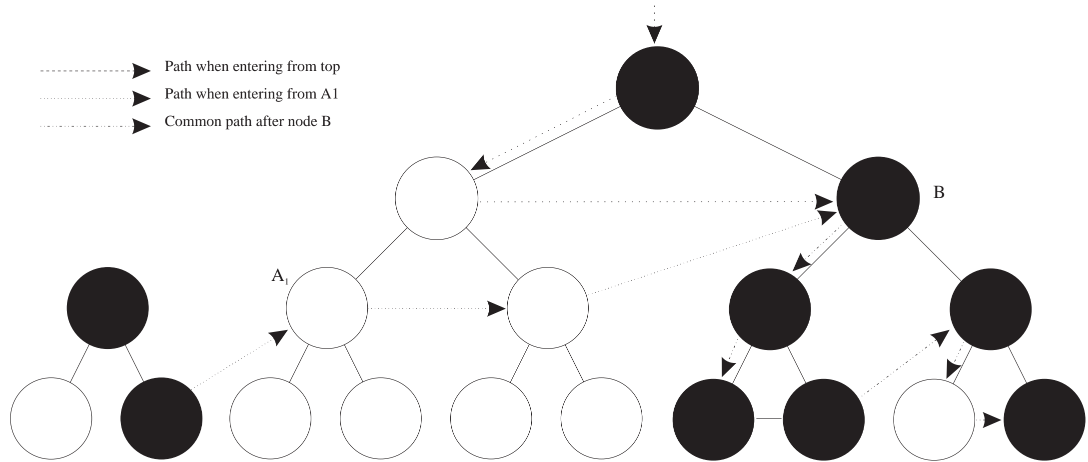


Figure 4.6: Possible paths through the right sub-HBV.

that point on they coincide. The point at which they first coincide determines how many extra nodes were visited (or how many visits were saved). For example, if the two paths coincide at A_1, then we have saved $h-1$ visits by entering from the side instead of the root. The problem now becomes determining when the two paths coincide at A_1, or, equivalently, when the path from the root will visit A_1. That path will visit A_1 exactly when one of the four leftmost leaves contains a 1, which occurs with probability $1-q^4$. So, with that probability, we save $h-1$ visits (or we do $1-h$ extra visits).

If node A_1 is not the point at which the two paths coincide then the four leftmost leaves contain 0's. Then the path that entered from the side eventually visits the first two nodes on level 1 and then the second node on level 2. Call that node B. It is possible that the path from the root visits B as well. That will happen if the four leftmost leaves contain 0's but the eight leftmost leaves have a 1 among them. That happens with probability $q^4(1-q^4)$. If that is the case then the path from the root visited $h-1$ nodes before arriving at B. The path from the side visited 2 nodes before getting to B, so with probability $q^4(1-q^4)$ we save

$h-3$ visits.

We can proceed in this manner. At each level i there is a unique node where the two paths can coincide (for $i>1$ that node is the second from the left). The two paths will coincide at that node with probability $(1-q^4)$ for $i=1$, $q^{2^i}(1-q^{2^i})$ for $i>1$. If the two paths concide at level i, then we will do $h-i-1$ visits from the top and i visits from the side before getting to the intersection.

It is also possible that the two paths do not coincide at all. This will be the case if all the leaves of the right sub-HBV contain 0's, which will occur with probability q^{2^h}. In that case, if we had entered from the root we would have visited only one node, the root. When entering from A_1 we have to visit h nodes, so we will have visited $h-1$ extra nodes.

Putting this all together, if we enter the right sub-HBV from node A_1 we will visit, on average,

$$(1-q^4)(1-h)+\sum_{i=2}^{h-1} q^{2^i}(1-q^{2^i})(2i-h+1)+q^{2^h}(h-1)$$

extra nodes.

Of course, we enter at A_1 only with probability $(1-q^2)$. Call the ancestors of A_1, in order from the parent to the root, $A_2,\ldots,A_h$. We may enter the right sub-HBV at any of those points as well. Denote by $Q(i)$ the probability that we enter the right sub-HBV at node A_i and by $R(i)$ the expected number of extra visits we do when entering there. When will we enter at A_2? When one of the six rightmost leaves of the left sub-HBV are 1's but the two rightmost are 0's. That happens with probability $(1-q^4)q^2$. When will we enter at A_3? When one of the fourteen rightmost leaves are 1's but all the six rightmost are 0's, which occurs with probability $(1-q^8)q^6$. In general,

$$Q(i)=(1-q^{2^i})q^{2^i-2}=q^{2^i-2}-q^{2^{i+1}-2}.$$

To figure out the expected number of extra visits we do when entering at node A_i for $i > 1$, we can go through the same sort of reasoning we did to figure out how many extra visits are done when entering at A_1. Doing so, we get

$$R(i,h) = (1 - q^{2^{i+1}})(i-h) + \sum_{j=i+1}^{h-1} q^{2^j}(1-q^{2^j})(2j-i-h) + q^{2^h}(h-i).$$

Which can be shown to equal (since the sum partially telescopes)

$$(i-h) + 2\sum_{j=i+1}^{h} q^{2^j}.$$

Note that for $h > 0$,

$$\begin{aligned} R(i,h-1) &= (i-h+1) + 2\sum_{j=i+1}^{h-1} q^{2^j} \\ &= 1 - 2q^{2^h} + (i-h) + 2\sum_{j=i+1}^{h} q^{2^j} \\ &= R(i,h) - 2q^{2^h} + 1. \end{aligned}$$

Now we can finally complete our recurrence for $E(i)$ with an expression for $d(h)$, which is

$$d(h) = \sum_{i=1}^{h-1} Q(i)R(i,h).$$

Remember that this goes in our recurrence for $E_0(h)$ in formula 4.1. Evaluating this recurrence for various values of p and h gives us the following table:

2^h / p	0.0020	0.0078	0.0312	0.1250	0.5000
2	1.00780	1.03113	1.12305	1.46875	2.50000
4	1.03119	1.12403	1.48462	2.76514	5.87500
8	1.09726	1.38429	2.46468	5.88345	12.0859
16	1.27125	2.05944	4.86843	12.3988	23.5389
32	1.70426	3.69724	10.2282	24.9916	45.4450
64	2.73428	7.43483	21.2403	49.2503	88.2571
128	5.08918	15.4689	42.8576	96.7687	172.881
256	10.2654	31.8338	85.1779	190.805	341.130
512	21.1784	64.1644	168.820	377.879	676.627
1024	43.3021	127.915	335.104	751.026	1346.62
2048	87.1522	254.417	666.672	1496.32	2685.61
4096	173.942	506.421	1328.81	2985.91	5362.59
8192	346.525	1009.43	2652.08	5964.09	10715.5
16384	690.689	2014.45	5297.63	11919.4	21420.4
32768	1378.02	4023.48	10587.7	23829.2	42829.3
65536	2751.67	8040.55	21166.9	47647.6	85645.9
131072	5497.99	16073.7	42324.3	95283.4	171278.
262144	10989.6	32138.9	84638.0	190554.	342542.
524288	21971.9	64268.5	169265.	381094.	685068.

One thing to notice about this table is that in each column, after some initially large increases, if we double the size of the tree we less than double the expected number of visits. The point at which this first occurs is sooner for denser sets. This suggests that, for a fixed p, the expected number of visits is $O(n)$, where the constant of proportionality depends on p. Estimating the constant from the last

row of the above table gives us

p	0.0020	0.0078	0.0312	0.1250	0.5000
visits	0.042n	0.123n	0.323n	0.726n	1.31n

We can prove that the recurrence for $E_0(h)$ is linear simply by noting that the number of bits examined is no more than the number of bits in the HBV, which is linear in n. To get a bound on the constant of proportionality for a fixed p, it suffices to find an h such that for $h' > h$ we have $E_0(h'+1) < 2E_0(h')$. We can find such an h if we can get a fix on $d(h)$, which we do in the following theorem.

Theorem 4.1 *For any fixed* p, $\lim_{h\to\infty} d(h) = -\infty$.

Proof. Applying the recurrence for $R(i,h)$ to the definition of $d(h)$ gives us

$$\begin{aligned} d(h) &= \sum_{i=1}^{h-1} Q(i)R(i,h) \\ &= \sum_{i=1}^{h-2} Q(i)R(i,h) + Q(h-1)R(h-1,h) \\ &= \sum_{i=1}^{h-2} Q(i)\left(R(i,h-1) + 2q^{2^h} - 1\right) + Q(h-1)R(h-1,h) \\ &= d(h-1) + \sum_{i=1}^{h-2} Q(i)(2q^{2^h} - 1) + Q(h-1)R(h-1,h). \end{aligned}$$

The sum involving $Q(i)$ telescopes, yielding

$$\begin{aligned} d(h) &= d(h-1) + (2q^{2^h})(1 - q^{2^{h-1}-2}) + (q^{2^{h-1}-2} - q^{2^h-2})(2q^{2^h} - 1) \\ &= d(h-1) + 2q^{2^h} - 2q^{2^{h+1}-2} + q^{2^h-2} - 1 \\ &= d(h-1) + (q^{2^h-2} - 1)(1 - 2q^{2^{h+1}}). \end{aligned}$$

Since $q = 1 - p < 1$,

$$\lim_{h\to\infty} (q^{2^h-2} - 1)(1 - 2q^{2^{h+1}}) = -1,$$

and therefore for large enough h, d is decreasing and in fact will decrease in ever larger amounts, which proves the result. □

Knowing that $d(h)$ can get arbitrarily small tells us that after a certain point $E_0(h)/2^h$ will be monotonically decreasing. We can then get an upper bound on $E_0(h)/2^h$ by finding a local maximum, which is what we did to produce table 4.4.1.

The procedure analyzed above is probably not the one that would come to mind first. A more natural approach would be to traverse the tree by a recursive, top-down descent. We can rewrite the next procedure to mimic that behavior:

```
procedure next'(integer n) returns integer
  t = nth leaf
  while t not root and (t is left child implies t's sibling unset)
    t = parent(t)
  end while
  if bit t is the root then
    return -1
  else
    while bit t is not a leaf
      if bit t's left child is set then
        t = leftchild(t)
      else
        t = rightchild(t)
      end if
    end while
    find i such that bit t is ith leaf at bottom of tree
    return i
```

```
  end if
end procedure
```

We can compute the expected number of bits examined for this procedure in order to compare it to our first one.

Let p be the probability that a given integer is contained in a set represented by an HBV of height h. Denote by $F(h)$ the expected number of bits examined by procedure next' when used to sort the set. Note that the root is always visited and a non-root node at level i is visited if and only if its parent (which is at level $i-1$) is non-zero, which happens with probability $1 - q^{2^{h-(i-1)}}$ where $q = 1 - p$. There are 2^i nodes at level i, so

$$F(h) = 1 + \sum_{i=1}^{h} 2^i (1 - q^{2^{h-(i-1)}}).$$

Compare the following table of values of $F(h)$ with the table for $E_0(h)$:

2^h / p	0.0020	0.0078	0.0312	0.1250	0.5000
2	1.00780	1.03113	1.12305	1.46875	2.50000
4	1.03119	1.12403	1.48462	2.76514	5.87500
8	1.09342	1.36969	2.41783	5.84306	12.7422
16	1.24842	1.97525	4.63225	12.4500	26.4843
32	1.61814	3.39443	9.54039	25.8721	53.9687
64	2.47149	6.57818	19.8186	52.7438	108.937
128	4.38576	13.4235	40.6028	106.488	218.875
256	8.55906	27.5784	82.2051	213.975	438.750
512	17.3831	56.1207	165.410	428.950	878.499
1024	35.4960	113.241	331.820	858.900	1758.00
2048	71.9556	227.482	664.641	1718.80	3517.00
4096	144.910	455.963	1330.28	3438.60	7034.99
8192	290.821	912.927	2661.56	6878.20	14071.0
16384	582.642	1826.85	5324.13	13757.4	28143.0
32768	1166.28	3654.71	10649.3	27515.8	56286.9
65536	2333.57	7310.41	21299.5	55032.6	112575.
131072	4668.13	14621.8	42600.0	110066.	225151.
262144	9337.27	29244.7	85201.0	220133.	450302.
524288	18675.5	58490.3	170403.	440268.	900606.

Note that for large, dense sets, the first procedure examines far fewer bits than the second, and for large enough sparse sets the first procedure examines fewer bits. Of course, each procedure does work in addition to the bit examinations. Measuring the total work done by applying each procedure to randomly generated sets shows that the point at which the first procedure outperforms the second

comes for smaller sets than indicated by the tables above.

We now compare the performance of this new structure to that of a binary tree used for the same purpose. Suppose that in our determinization algorithm we perform m_1 insertions, m_2 of which are distinct. After the insertions we do several equality tests, each of which can be done with a sort. Since there are no more insertions after the first equality test, we can remember the result of the first sort and use it again instead of recomputing it. Essentially, we are doing m_1 insertions and one sort for each set.

A binary tree will need on average $O(m_1 \log m_2)$ work to do the insertions followed by a trivial amount of work to sort the set. If $m_1 = m_2 = pn$ then that is equivalent to $O(n \log n)$. The HBV, on the other hand, does only $O(n)$ work for the insertions and $O(n)$ work for the sort for a total of $O(n)$. If the insertions are not distinct (that is, if $m_1 \neq m_2$) then the HBV is yet faster since each duplicate insertion costs the binary tree $O(\log m_2)$ time but only costs the HBV $O(1)$ time. We can then conclude that for each p there is an n after which the HBV will outperform a binary tree. For small p that n is quite large, larger than we have needed in the tests we have run so far. But for large p the break even point comes for quite small n (for example, for $n = 1024$ and $p = .25$ the HBV outperforms the binary tree by a factor of about two).

However, as p increases, the advantage the HBV has over the bit vector gets smaller and smaller until it disappears completely. Tests have shown that for $p < \frac{1}{512}$ the HBV outperforms an ordinary bit vector. However, for such sparse sets the range must be high indeed for the HBV to outperform a binary tree. We need to improve our structure a bit.

4.4.2 8-ary Hierarchical Bit Vectors

The high performance of the ordinary bit vector comes from the fact that no processor works one bit at a time. If we want to list the elements stored in a bit vector in order using a 32-bit processor, we can break the vector into words of 32 bits each and examine each word using one instruction. If a word is zero then we can ignore it. Only if it is non-zero do we have to slow down and check it bit by bit. We use the same sort of strategy to speed up our HBV's.

An 8-ary HBV ($8HBV$ for short) is similar to an HBV, but each node in the tree has eight children. Insertions and membership tests can be done in much the same way as in the original HBV. The next operation can be reworked to take advantage of the power of microprocessors to examine more than one bit at a time. In the procedure below, $BYTE(n)$ refers to the group of eight bits containing bit n.

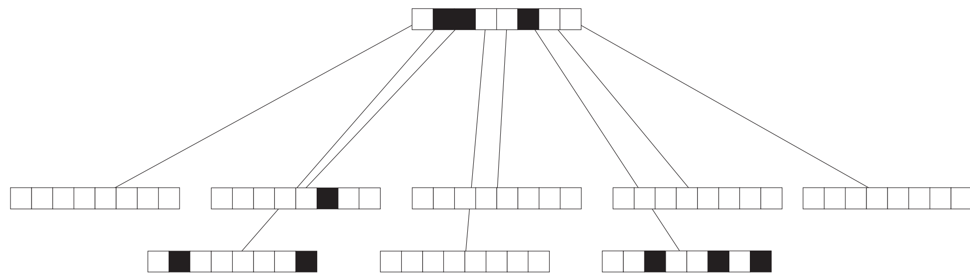


Figure 4.7: An $8HBV$ representing $\{9, 15, 42, 45, 47\}$.

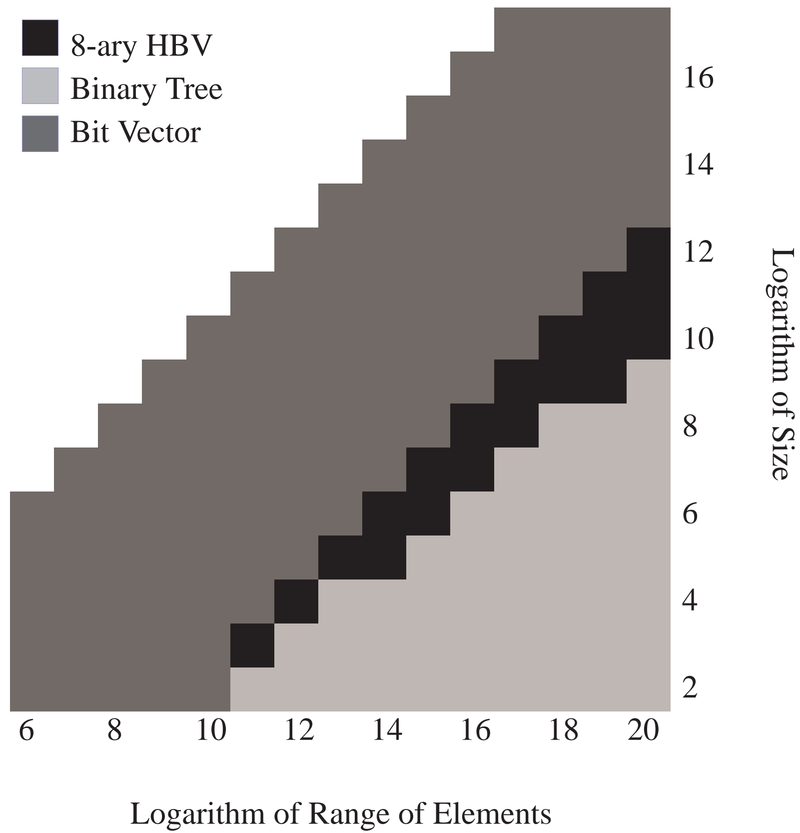
```
procedure next(n) returns integer
  y = byte containing (n+1)th leaf
  t = index of bit in y corresponding to (n+1)th leaf
  while y is not root and t low order bits of y are clear
    t = y & 7
```

```
    y = y / 8
  end while
  y = y * 8 + index of MSB after t set in y
  while y is not a leaf
    y = y * 8 + index of MSB set in y
  end while
  find i such that MSB set in y corresponds to ith leaf
  return i
end procedure
```

Here the idea is that if we want to start at leaf n and find the next bit that is set, we first check the siblings to the left of n all at the same time. If none are set, we go to the parent of n and check all of its siblings simultaneously. We proceed up and across the tree in this manner until we find a set bit, then proceed down the tree to find the first set leaf under that bit.

While the original HBV beats the oridinary bit vector for p lower than about $\frac{1}{512}$, the new $8HBV$ beats the bit vector for p up to about $\frac{1}{128}$ and beats the binary tree for large sets. We summarize this information in a chart that shows, for m-subsets of $\{1, \ldots, n\}$, which structure had a better total time for initialization, insertions, and sort. In doing these tests, sorts were done only once and no duplicate insertions were attempted. The more duplicate insertions that are done, the better the bit vector and HBV structures will be. The more sorts that are done, the better the binary tree will be. The data indicate, as expected, that bit vectors perform well for dense sets and that binary trees work well for small sets. There is also a narrow in-between range for which the HBV is the best performer.

Each trial does only one sort, but in determinizing a DFA we may have to



compare the same state to others many times. Sets are compared by sorting, so many comparisons will lead to many sorts. However, the trials are still relevant since all insertions into our sets are done before any sorts. Therefore, we can sort one and use the results of that sort for future operations.

Chapter 5

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 lead to unmanagably large automata. 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 interesting property of certain automata.

5.1 Trap State Reduction

Suppose we have a formula f of the form $g(\chi_1, \ldots, \chi_n) \wedge h(\chi_1, \ldots, \chi_n)$ 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 although it is extremely helpful to use this method of partial minimization, it alone will not solve the problem of large machines.

5.2 Algorithms for Minimizing Arbitrary DFAs

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 [4], 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 [10]. 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, since Hopcroft's algorithm is hurt by the fact that it needs to compute the inverse transition relation, which is done in $O(|\Sigma| n)$ time using our internal representation of automata. We need no preprocessing to help the standard algorithm with our current representation. Analyzing the time spent in the minimization algorithm when running our 200 hard inputs suggests that Hopcroft's algorithm is better for machines with over 300 states.

5.3 Layered Automata

Finally, we can exploit the special structure of some of our automata to use a new algorithm that runs in $O(|\Sigma| n)$ time. The algorithm is presented below. The special requirement is that the automaton has no non-trivial cycles. This notion is formalized below.

Definition 5.1 *A deterministic finite automaton $M = (K, \Sigma, \delta, s, F)$ is said to be layered if whenever $(q, w) \vdash_M^* (q, e)$ for a non-empty string $w_1 \dots w_k$ we have $\delta(q, w_1) = \cdots = \delta(q, w_k) = q$.*

All of the automata generated for atomic formulas have this property. Furthermote, it is preserved through the operations of complementation, union, and intersection.

Theorem 5.2 *If $G = (K_G, \Sigma, \delta_G, s_G, F_G)$ and $H = (K_H, \Sigma, \delta_H, s_H, F_H)$ are layered automata, then*

(a) $\bar{G} = (K_G, \Sigma, \delta_G, s_G, K_G - F_G)$*;*

(b) $(G \times H) = (K_G \times K_H, \Sigma, \delta_{G\times H}, (s_G, s_H), F_{G\times H})$ *where* $F_{G\times H} \subseteq G \times H$*, and* $\delta_{G\times H}(q_G, q_H) = (\delta_G(q_G), \delta_H(q_H))$*; and*

(c) G' *where* G' *is minimal and equivalent to* g

are all layered.

Proof.

(a) Is clear: changing the final states has no effect on the structure of the transitions.

(b) Let G, H, and $G \times H$ be as given and suppose there is a nontrivial cycle in $G \times H$. Then we would have $((q_G, q_H), uv) \vdash^*_{G\times H} ((q'_G, q'_H), v) \vdash^*_{G\times H} ((q_G, q_H), e)$ where $u \in \Sigma^*$ is non-empty and $(q'_G, q'_H) \neq (q_G, q_H)$. Assume, without loss of generality, that $q'_G \neq q_G$. Then, by the choice of $\delta_{G\times H}$, we would have $(q_G, u) \vdash^*_G (q'_G, e)$, which contradicts the assumption that G is layered.

(c) . Let G and G' be as given and suppose that G' is not layered. Then there is a non-empty string $w \in \Sigma^*$ such that $(q, w) \vdash^*_{G'} (q, e)$. Find the state q' such that $(q, w) \vdash^*_G (q', e)$. Since G and G' are equivalent, q' must be equivalent to q (it cannot equal q since G is layered). But then if $(q', w) \vdash^*_G (q'', e)$ we must have q'' equivalent, but not equal, to q and q'. Continuing, we would get an infinite sequence of distinct equivalent states in G which is impossible because G has only a finite number of states.

□

Unfortunately, existential quantifiers can destroy the structure (Figure 5.1).

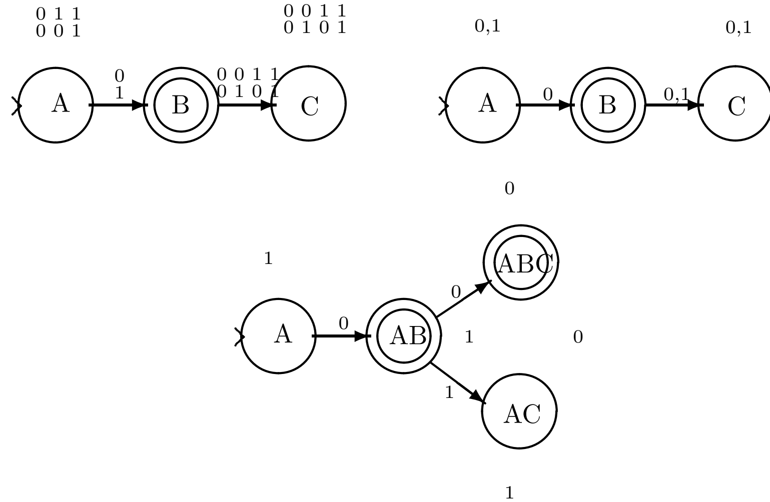


Figure 5.1: Layers are not preserved by nondeterminism

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 keeps track of three sets of states. One set contains states that have already been minimized, which we will denote C (for civilization; these are the states that have been put into good order). Another set contains the states that are currently being worked on; these will be called F (for frontier; these are the states currently being tamed). The last set is for everything else and will be called W (for wilderness). For each state q we will assign a level which we denote $l(q)$. The algorithm follows:

(1) Find all states that have no transitions out of them. Merge all such final states into one state a and all such non-final states into another state r. Set $l(a) = 0$, $l(r) = 0$ and let $C = \{a, r\}$, $F = \emptyset$, and $W = Q - C$.

(2) Let $F = F \cup \{q \in W | (\forall \sigma \in \Sigma)[\delta(q, \sigma) \in C]\}$ and $W = W - F$.

(3) While there exists a state $q \in F$ such that q is equivalent to some state r in C,

   (a) merge q with r, let $F = F - \{q\}$;

   (b) let $F = F \cup \{q \in W | (\forall \sigma \in \Sigma)[\delta(q, \sigma) \in C]\}$ and let $W = W - F$.

(4) Merge equivalent states in F, for each $q \in F$ let $l(q) = L + 1$, let $L = L + 1$, $C = C \cup F$, and $F = \emptyset$. If $W \neq \emptyset$ then go back to step 2.

The following theorem establishes the correctness of the algorithm by showing that states in C are never equivalent to each other; it also shows how to find equivalent states to merge in steps 2 and 3.

Theorem 5.3 *After each step of the algorithm, the following are true:*

(a) For any integer i such that $0 \leq i \leq L$ there exists a $c \in C$ such that $l(c) = L$.

(b) For each state $q \in C \cup F$ and each $\sigma \in \Sigma$, $\delta(q, \sigma) \in C$.

(c) For any pair of states q and r in C, if $\delta(q, \sigma) = r$ for some $\sigma \in \Sigma$ then $l(q) > l(r)$.

(d) For all $q \in F$ there is a symbol $\sigma \in \Sigma$ such that $l(\delta(q, \sigma)) = L$.

(e) For all states $q \in F$ if there are transitions from q into two different states r and s in C, with $l(r) = L$, and $l(s) = L$ then q is equivalent to neither of those states.

(f) For all states $q \in F$ and $r \in C$, q is equivalent to c if and only if they are both final or both non-final and $l(r) = L$, and for all symbols $\sigma \in \Sigma$ either $\delta(q, \sigma) = \delta(r, \sigma)$ or both $\delta(q, \sigma) = q$ and $\delta(r, \sigma) = r$.

(g) *For any pair of states q and r in F, q and r are equivalent if and only if they are both final or both non-final and for each $\sigma \in \Sigma$, $\delta(q,\sigma) = \delta(r,\sigma)$ or $\delta(q,\sigma) = q$ and $\delta(r,\sigma) = r$.*

(h) *No two states in C are equivalent.*

Proof. (a)-(h) certainly hold after step 1. Suppose (a)-(h) hold before a certain step in the algorithm. We need to show that they still hold after that step executes.

(a) The only step that can affect this is step 4, but while it increases L by 1 it also puts all elements of F into C and assigns them to level L.

(b) Clear from the choice of F in step 2.

(c) Merges won't affect this statement since the state eliminated by the merge is always in F, and no state in C has a transition to a state in F by (b). Step 4 will preserve the truth of this as well since beforehand all states in F must have transitions only into C and hence only to states q with $l(q) \leq L$. Each state r in F is then given $l(r) = L + 1$.

(d) Suppose $q \in F$ but $\max_{\sigma\in\Sigma} l(\delta(q,\sigma)) = L' < L$. Consider the execution of the algorithm after the L'th execution of step 4. Control goes back to step 2 at which point q would be put into F. We must remove q F in step 3 or step 4; both would occur before L is increased past L'.

(e) Suppose $q \in F$, and there exist symbols $\sigma_1, \sigma_2 \in \Sigma$ and states $r, s \in C$ such that $\delta(q,\sigma_1) = q$, $\delta(q,\sigma_2) = s$, $l(r) = l(s) = L$ and $r \neq s$. Suppose further that $q \equiv r$. Then $s = \delta(q,\sigma_2) \equiv \delta(r,\sigma_2)$. Since C is minimized we must have $\delta(r,\sigma_2) = s$, which contradicts (c).

(f) The if direction is clear. Suppose $q \in F$ is equivalent to some $r \in C$. Clearly q and r must be both final or both nonfinal. We must show that $l(r) = L$ and that the two states agree on all symbols.

Suppose $l(r) < L$. By (d) we know there is an $s \in C$ and a symbol $\sigma \in \Sigma$ such that $l(s) = L$ and $\delta(q, \sigma) = s$. Since $q \equiv r$ we must also have $s = \delta(q, \sigma) \equiv \delta(r, \sigma)$, which is in C by (b). Since C is minimized we then have $s = \delta(r, \sigma)$, which contradicts (c).

Finally, suppose there is a symbol $\sigma \in \Sigma$ such that $\delta(q, \sigma) \neq \delta(r, \sigma)$ and either $\delta(q, \sigma) \neq q$ or $\delta(r, \sigma) \neq r$. If $\delta(q, \sigma) \neq q$ and $\delta(q, \sigma) \neq \delta(s, \sigma)$ then those must be two different states in C and hence not equivalent, so q and r couldn't be equivalent. If $\delta(r, \sigma) \neq r$ and $\delta(q, \sigma) \neq \delta(r, \sigma)$ then those two states are either different (hence inequivalent) states in C or $\delta(q, \sigma) = q$. In that case, since we must have $l(\delta(r, \sigma)) < l(r)$ we would have q equivalent to a state s with $l(s) < L$, which the previous paragraph tells us can't happen.

(g) Again, the if direction should be clear. The only if direction is similar to case (f).

(h) Follows from (f) and (g).

The algorithm runs in $O(\Sigma n)$ time, where n is the number of states in the machine. To show this, we charge work to the transitions in the original machine. Figuring out which states belong to F can be done by first computing the inverse transition relation. Then, when a state is moved into C, we examine the inverse transitions out of it, incrementing a counter for the state at the other end. When a state's counter has reached $|\Sigma|$ it is moved into F. The process of computing the inverse transition relation examines each transition once and each inverse transition

(hence transition) causes a counter to be incremented only once.

Each state needs to be checked for equivalence against states in the previous layer. The theorem tells us that there is at most one candidate for equivalence in the previous layer for each state in F. Therefore, each state will only have to check its transitions against states in previous layers once. A given state in a previous layer may be checked against several members of F, but if we charge that work to the state in F we see that only two more examinations are done for each transition.

Each state in F also needs to be checked for equivalence against other states in F. That is done with bucket sort, which examines each transition once. The total work done is constant for each transition and so is linear in the number of transitions, $|\Sigma|n$.

Chapter 6

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 [7] 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 is 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.

For any term t we can now construct a $WS1S$ formula $f(X, z, t)$ that expresses "the set X represents term t" (where z is some free variable we are given to use). The basic forms are as follows:

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

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

where

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

and

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

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

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

$$
\begin{aligned}
f(X,z,t_1+t_2) = \quad & \forall(T_1,T_2)([f(T_1,z,t_1) \wedge f(T_2,z,t_2)] \rightarrow (\\
& (0 \notin T_1 \wedge 0 \notin T_2 \wedge 0 \notin X \wedge plus(T_1,T_2,X)) \quad \vee \\
& (0 \in T_1 \wedge 0 \in T_2 \wedge 0 \in X \wedge plus(T_1,T_2,X)) \quad \vee \\
& (0 \in T_1 \wedge 0 \notin T_2 \wedge bigger(T_2,T_1) \wedge 0 \notin X \wedge minus(T_2,T_1,X)) \quad \vee \\
& (0 \in T_1 \wedge 0 \notin T_2 \wedge bigger(T_1,T_2) \wedge 0 \in X \wedge minus(T_1,T_2,X)) \quad \vee \\
& (0 \in T_1 \wedge 0 \notin T_2 \wedge same(T_1,T_2) \wedge f(X,z,0)) \quad \vee \\
& (0 \notin T_1 \wedge 0 \in T_2 \wedge bigger(T_1,T_2) \wedge 0 \notin X \wedge minus(T_1,T_2,X)) \quad \vee \\
& (0 \notin T_1 \wedge 0 \in T_2 \wedge bigger(T_2,T_1) \wedge 0 \in X \wedge minus(T_2,T_1,X)) \quad \vee \\
& (0 \notin T_1 \wedge 0 \in T_2 \wedge same(T_1,T_2) \wedge f(X,z,0)) \quad))
\end{aligned}
$$

where

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

and finally $carry(X,Y,Z,C,z)$ 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 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.

Chapter 7

Conclusion

We have studied many aspects of the $WS1S$ decision procedure. We have determined several data structures that work well with it. We have introduced another structure which does not currently help with the decision procedure but may be helpful in other applications. We have also suggested new methods of minimizing automata and shown that a previously known one does not work terribly well (at least on the automata that come up in our work).

Several groups are working on implementations of Presburger Arithmetic to solve various problems in compilers that optimize loops for parallel procesing or in verifying properties of other finite machines. We, on the other hand, use Presburger Arithmetic only to give us formulas on which to test our algorithms and data structures and do not expect the decision procedure for $WS1S$ to be competitive with the specialized algorithms for Presburger Arithmetic (and indeed it isn't).

Some groups are working on similar projects. Klarlund, *et al.* have developed `MONA`, which implements the decision procedures for $WS1S$ and $WS2S$. Their approach and motivation are different from ours: they use BDD's to represent their automata and their goal is closer to that of those who implement Presburger Arithmetic.

Future work in this area could include a more theoretical, rather than empirical, analysis of the structures and algorithms used. For example, it would be good to have a theorem that relates the size of machines under one encoding to the size under another. It would also be good to analyze our n-ary tRIe structure more thoroughly. Finally, we could apply our approach to $W2S2$ in hopes of finding new structures and algorithms there as well.

BIBLIOGRAPHY

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

[2] Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Basic Books Inc., 1962.

[3] D. Hilbert. Mathematische probleme. *Archiv der Mathematik und Physik*, 3, 1901.

[4] John 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.

[5] 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.

[6] Albert 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.

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

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

[9] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science.* MIT Press, 1990.

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