Spring 5-1-2018

Minimization Techniques for Symbolic Automata

Jonathan Homburg
jonhom1996@gmail.com

Recommended Citation

Homburg, Jonathan, "Minimization Techniques for Symbolic Automata" (2018). *Honors Scholar Theses*. 565.
https://opencommons.uconn.edu/srhonors_theses/565

Minimization Techniques for Symbolic Automata

Jonathan Homburg
Honors Thesis
Supervised by Parasara Sridhar Duggirala

Abstract

Symbolic finite automata (SFAs) are generalizations of classical finite state automata. Whereas the transitions of classical automata are labeled by characters from some alphabet, the transitions of symbolic automata are labeled by predicates over a Boolean algebra defined on the alphabet. This allows for SFAs to be efficiently constructed over extremely large, and possibly infinite, alphabets. This thesis examines an existing incremental algorithm for the minimization of deterministic finite automata. Several extensions of this algorithm to deterministic symbolic automata are introduced. Although more efficient algorithms already exist for deterministic SFA minimization, the presented algorithms are uniquely designed to minimize an automaton incrementally. That is, the algorithms may be halted at any time, for computations to be run on a partially minimized SFA that accepts the same language as the input, and then resumed later.

Contents

Chapter 1

Introduction

Symbolic finite automata extend classical automata by removing the assumption of a finite alphabet set. Instead, the alphabet of a symbolic automaton is defined by a Boolean algebra which may have an infinite domain. To accommodate this alphabet, the state transitions within the symbolic automaton are labeled by predicates over the Boolean algebra rather than by characters from the domain. An overview of the theory and applications of symbolic automata is provided in [8].

The minimization of deterministic symbolic finite automata has already been studied in [16] and [6]. In particular, D'Antoni and Veanes [6] extend two classical algorithms for DFA minimization, Moore's algorithm and Hopcroft's algorithm, to deterministic symbolic automata. However, these algorithms both work by partitioning an automaton's set of states and progressively refining until all states within a block of the partition are indistinguishable. When this process terminates, a minimal automaton is obtained by collapsing each block of the partition into its own state. Although potentially efficient, the entire computation of the algorithm needs to be completed before an automaton with an equivalent language to the input can be returned.

An algorithm for the incremental minimization of DFAs has been presented by Almeida et al. [2]. Unlike Moore's algorithm or Hopcroft's Algorithm, this incremental algorithm does not use partition refinement. Instead, it iteratively checks for equivalence on pairs of states within the DFA and combines the states if they are found to be equivalent. This process continues until the automaton with the minimum number of states and equivalent language to the input has been reached, at which point the algorithm terminates. Thus, this algorithm has the benefit that it can be halted at any time and return a DFA with a smaller state space than the input automaton and an equivalent language.

This thesis is concerned with extending the incremental approach to the minimization of deterministic SFAs. Chapter 2 will provide background knowledge on the definition of symbolic automata as well as relevant results. Chapter 2 will also provide an overview of the incremental DFA minimization algorithm given in [2]. This algorithm will be extended to symbolic automata in three different ways in Chapter 3. Specifically, a naive algorithm, based on minterm generation, and an efficient algorithm, that takes advantage of the symbolic setting, will be introduced. Additionally, a novel, but ultimately inefficient, incremental algorithm based on "dependency checking" will be presented. The implementation of these algorithms will be described in Chapter 4 and their performance will be experimentally evaluated in Chapter 5. Lastly, Chapter 6 will review related work and suggest possible avenues of future research.

Chapter 2

Preliminaries

2.1 Symbolic Automata

The following definitions are predominantly adapted from [6].

Definition 1. An *effective Boolean Algebra* $\mathcal{A}$ is a tuple $(\mathcal{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathcal{D}$ is a recursively enumerable (r.e.) set of domain elemenets, Ψ is an r.e. set of predicates closed under $\vee, \wedge, \neg$, and with $\bot, \top \in \Psi$. $[\![\_]\!] : \Psi \to 2^{\mathcal{D}}$ is the *denotation function*, an r.e. function with $[\![\bot]\!] = \varnothing$, $[\![\top]\!] = \mathcal{D}$ and for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\psi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\psi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathcal{D} \setminus [\![\psi]\!]$. If $[\![\varphi]\!] \neq \varnothing$ for $\varphi \in \Psi$, we write $IsSat(\varphi)$ and say that φ is satisfiable.

In our expirements, we exclusively use a Boolean algebra whose domain consists of bit vectors of length $k = 16$, with each unique bit vector corresponding to a unique symbol in the unicode character set. Predicates over this domain can be represented as binary decision diagrams (BDD) of depth k with Boolean operations corresponding to operations on BDDs [6]. Alternatively, predicates can be represented as Boolean formulas with Boolean operations corresponding to calls to a SAT solver. In practice, an effective Boolean algebra can be thought of as an API with methods implementing Boolean operations [8].

Boolean algebras will define the input alphabet of symbolic automata. Inputs will be strings in $\mathcal{D}^*$, finite sequences of elements from $\mathcal{D}$, and transitions will be labeled by predicates in Ψ.

Definition 2. A *symbolic finite automaton* (SFA) M is a tuple $(\mathcal{A}, Q, q^0, F, \Delta)$ where $\mathcal{A}$ is an effective Boolean algebra (and is called the alphabet of M), Q is a finite set of states, q^0 is the initial state, $F \subseteq Q$ is the set of accepting (or final) states, and $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is a finite set of transitions.

A transition $\rho = (q, \varphi, q') \in \Delta$ will also be denoted by $q \overset{\varphi}{\to} q'$. ρ is called feasible if φ is satisfiable. For $a \in \mathcal{D}_{\mathcal{A}}$, an a-transition is a transition $q \overset{\varphi}{\to} q'$ with $a \in [\![\varphi]\!]$. In this case, the a-transition will also be denoted $q \overset{a}{\to} q'$.

Definition 3. Let $M = (\mathcal{A}, Q, q^0, F, \Delta)$ be an SFA. A string $w = a_1 a_2 \ldots a_k \in \mathcal{D}^*_{\mathcal{A}}$ is *accepted* at state $p \in Q$ if there exists a transition $p_{i-1} \overset{a_i}{\to} p_i$ in M for $1 \leq i \leq k$ with $p_0 = p$ and $p_k \in F$. This is denoted $w \in L_p(M)$. The *language accepted by* M is $L_{q^0}(M)$ and denoted $L(M)$.

Any classical finite automata can be modeled as an SFA. To see this, let N be a (possibly non-deterministic) finite automata. Define an effective Boolean algebra $\mathcal{A}$ as the equality algebra over $\mathcal{D}_{\mathcal{A}} := \Sigma_N$. That is, the set of predicates $\Psi_{\mathcal{A}}$ is the Boolean closure of the set of atomic predicates given by $\{\varphi_a \mid a \in \mathcal{D}_{\mathcal{A}}, [\![\varphi_a]\!] = \{a\}\}$ [8]. Lastly, let $\Delta_M := \{(p, \varphi_a, q) \mid p, q \in Q_N, a \in \Sigma_N \text{ and } (p, a, q) \in \Delta_N\}$. Under this construction, $M = \{\mathcal{A}, Q_N, q_N^0, F_N, \Delta\}$ is a symbolic finite automata with an equivalent language to N.

Some interesting properties of an SFA $M = (\mathcal{A}, Q, q^0, F, \Delta)$ are given by the following definitions:

- M is *deterministic* if for all transitions $p \xrightarrow{\varphi} q$ and $p \xrightarrow{\varphi'} q'$ in Δ, we have $IsSat(\varphi \wedge \varphi')$ implies $q = q'$.
- M is *complete* if for all states $p \in Q$ and all domain characters $a \in \mathcal{D}_{\mathcal{A}}$, there exists an a-transition out of p.
- M is *clean* if for all $p \xrightarrow{\varphi} q$ in Δ, p is reachable from q^0 and φ is satisfiable.
- M is *normalized* if for all $p, q \in M$ there exists at most one transition from p to q.
- M is *minimal* if M is deterministic, clean, normalized and for all $p, q \in Q$, $L_p(M) = L_q(M)$ implies $p = q$.

Similarly to classical automata, for an arbitrary SFA M, it is always possible to find a deterministic, complete, clean, or normalized SFA that accepts the same language as M. Determinization is outlined in [16] and completion, normalization and cleaning are outlined in [6]. If M is a deterministic and complete SFA, we define a transiton function $\delta_M : Q \times \mathcal{D}_{\mathcal{A}} \to Q$ such that $\delta_M(q, a) = q'$ where $q \xrightarrow{a} q'$ is a transition in M. For the remainder of this thesis, it will typically be assumed that a given SFA M is deterministic, complete, clean, and normalized unless stated otherwise. As with classical automata, minimal SFAs are unique (up to the renaming of states and equivalence of predicates). This is an immediate corollary to Theorem 1 in [6].

Definition 4. For an SFA $M = (\mathcal{A}, Q, q^0, F, \Delta)$, states q and q' within Q are *equivalent*, denoted $q \equiv_M q'$, if and only if $L_q(M) = L_{q'}(M)$.

$\equiv_M$ is an equivalence relation. So, for any $X \subseteq Q$, we can consider $X_{/\equiv_M} = \{q_{/\equiv_M} \mid q \in X\}$ where $q_{/\equiv_M}$ is the equivalence class containing state q. From this, we can define a new SFA $M_{/\equiv_M} := (\mathcal{A}, Q_{/\equiv_M}, q^0_{/\equiv_M}, F_{/\equiv_M}, \Delta_{/\equiv_M})$ where

$$\Delta_{/\equiv_M} := \left\{ \left(p_{\equiv_M}, \bigvee\nolimits_{(p,\varphi,q)\in\Delta} \varphi, q_{/\equiv_M} \right) \mid p, q \in Q; \exists \varphi \in \Psi \text{ with } (p, \varphi, q) \in \Delta \right\}$$

Theorem 1 (Theorem 2 from [6])**.** *For a clean, complete, and deterministic SFA M, $M_{/\equiv_M}$ is minimal and $L(M_{/\equiv_M}) = L(M)$*

Definition 5. For an SFA M and state $q \in Q$, define the minimum distance from q to an accepting state as

$$dist(q) := \begin{cases} \min_{w \in L_q(M)} |w| & \text{if } L_q(M) \neq \varnothing \\ \infty & \text{otherwise} \end{cases}$$

where $|w|$ is the length of the string w.

It can immediately be seen that for $p, q \in Q_M$, if $dist(p) \neq dist(q)$ then $L_p(M) \neq L_q(M)$. In particular, if $dist(p) \neq dist(q)$, then $p \not\equiv_M q$. Our algorithms will take advantage of this fact.

2.2 Union-Find

The disjoint set, or union-find, data structure represents collections of distinct elements organized into dijoint sets. The following operations are defined on this data structure:

1. `Make`(i) creates a new set containing only the element i.
2. `Find(`i`)` returns a unique identifier for S_i, the set containing i. This identifier should be consistent in the sense that `Find`(j) should be equivalent for all elements $j \in S_i$.
3. `Union`(i, j) creates a new set S_k such that $S_k = S_i \cup S_j$ and sets S_i, S_j are destroyed.

If n is the number of elements in the disjoint set data structure, then any arbitrary sequence of $m > n$ `Find` operations and $n - 1$ `Union` operations can be implemented to perform in $O(m\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function. Because of the slow rate of growth of $\alpha(n)$, this runtime can be treated in practice as $O(m)$ runtime. This follows from the fact that for all practical inputs n, $\alpha(n) \leq 3$ [2, 14]. So, this runtime can be treated as $O(m)$ in practice. As in [2], the incremental algorithms presented within this thesis will use a disjoint sets data structure to build the equivalence relation $\equiv_M$ given in Definition 4.

2.3 Incremental Minimization of DFAs

In [2], Almeida et al. present an algorithm for the incremental minimization of deterministic finite automata. This algorithm is reproduced in Algorithm 1.

Let $D = (Q, \Sigma, \delta, q^0, F)$ be a DFA and let $n = |Q|$. Intuitively, this algorithm works by iterating through all pairs of states $(p, q) \in Q \times Q$ and exhaustively searching for a witness that p is not equivalent to q. If no such witness is found, then p is indistinguishable from q and the two states can be merged without affecting the language of D. When all such pairs are checked, merging all pairs of states found to be indistinguishable will return a minimal automaton. Alternatively, the indistinguishable pairs can be merged at any point in the runtime of the algorithm to produce a partially minimized DFA that accepts the same language as the input automaton. This is why the algorithm is referred to as incremental.

Initially, a disjoint set data structure is created containing n singleton sets, each containing a unique state from Q. Each set in this structure will contain all states that the algorithm has found to be indistinguishable. This structure is particularly useful because it maintains the transitive closure of equivalence classes. The algorithm also starts by initializing a set *neq* to contain all pairs of accepting and non-accepting states. This set will contain all pairs of states proven by the algorithm to not be equal. This is done in order to prevent repeating computation. We then continue by iterating through the pairs of states (p, q) and testing for equivalence by calling `isEquiv(`p, q`)`, given in Algorithm 2. By the commutativity of state equivalence, we need not actually check for the equivalence of all states. That is, if we already know whether or not p is equivalent to q, we need not check if q is equivalent to q. So, we assume that there exists a total ordering on the pairs of states, which can be simply achieved by labeling each state by a unique integer. Following this, we need only check for equivalence on all normalized pairs of states (p, q) such that $p < q$. The `Normalize` function takes a pair of states as input and returns this normalized form. Lastly, `JoinStates` merges all states that share the same equivalence class (that is, the same disjoint set).

```
1  Function IncrementalMinimizeDFA($D = (Q, \Sigma, \delta, q^0, F)$):
2      for $q \in Q$ do
3          Make($q$)
4      $neq = \{$Normalize($p, q$) $\mid p \in F, q \in Q \setminus F\}$
5      for $p \in Q$ do
6          for $q \in \{x \in Q \mid x > p\}$ do
7              if $(p, q) \in neq$ then
8                  continue
9              if Find($p$) = Find($q$) then
10                 continue
11             $equiv, path = \varnothing$
12             if isEquiv($p$,$q$) then
13                 for $(p', q') \in equiv$ do
14                     Union($p$',$q$')
15             else
16                 for $(p', q') \in path$ do
17                     $neq = neq \cup \{(p', q')\}$
18     return JoinStates($M$)
```

Algorithm 1: Incremental minimization of DFAs, reproduced from [2]

Before the call to `isEquiv`, global sets *equiv* and *path* are defined and initialized to $\varnothing$. *equiv* will contain all pairs of states whose equivalence we've tested. *path* tracks the path from the pair (p, q) in the initial `isEquiv` call to the pair in the current recursive call in the tree associated with the depth-first search traversal of the pairs of states in D.

`isEquiv` proceeds by, assuming $(p, q) \notin neq$ in which case we would immediately know the pair to be inequivalent, recursively calling `isEquiv` on all pairs of states (p', q') such that for some $a \in \Sigma$, $p' = \delta(p.a)$ and $q' = \delta(q, a)$. Because D is deterministic, there exists exactly one such pair for each $a \in \Sigma$. If we already know that p' is equivalent to q', or $(p', q') \in equiv$ (in which case, we've already tested for the equivalence of (p', q')), we need not actually make the recursive call because we can presume it will return true. We can also presume equivalence if $(p', q') \in path$, which indicates a cycle of equivalence tests has been reached. We only return false if we come across a pair $(p', q') \in neq$ which tells us that p' is necessarily distinguishable from q'. In fact, this pair is a witness to the nonequivalence of all pairs of states in *path*. In this case, on line 17 in Algorithm 1, we add all pairs in *path* to *neq*. Otherwise, if no witness is found and all calls to `isEquiv` terminate, then all pairs of states that were tested and stored in *equiv* are indistinguishable and can be merged (as done on line 14 of Algorithm 1).

```
1  Function isEquiv($p, q$):
2      if $(p, q) \in neq$ then
3          return False
4      if $(p, q) \in path$ then
5          return True
6      $path = path \cup \{(p, q)\}$
7      for $a \in \Sigma$ do
8          $(p', q') =$ Normalize(Find($\delta(p, a)$), Find($\delta(q, a)$))
9          if $p' \neq q'$ and $(p', q') \notin equiv$ then
10             $equiv = equiv \cup \{(p', q')\}$
11             if not isEquiv($p', q'$) then
12                 return False
13     $path = path \setminus \{(p', q')\}$
14     $equiv = equiv \cup \{(p, q)\}$
15     return True
```

Algorithm 2: Equivalence check for states p and q in DFA D, reproduced from [2]

A full proof of this algorithm is provided in [2]. The authors of [2] also give the runtime of `IncrementalMinimizeDFA` as $O(kn^2\alpha(n))$ where $k = |\Sigma|$ and $n = |Q|$. However, this runtime is based on the assertion that ever pair of states recursively passed as arguments to `isEquiv` will not be considered again within the main loop of `IncrementalMinimizeDFA` (Lemma 5 in [2]). This is not true in general. As a counter-example, consider the case where `isEquiv` is first called on the pair of states (u_0, v_0) when `IncrementalMinimizeDFA` is run on the DFA given in Figure 2.1 with $\Sigma = \{0, 1\}$. If `isEquiv(`u_0, v_0`)` first checks for the equivalence of successor states given by the symbol $a = 0$, then recursive calls will be made on (u_1, v_1) and (u_2, v_2). Both of these recursive calls will return true because u_1 and u_2 are, in fact, equivalent to v_1 and v_2, respectively. Additionally, (u_1, v_1) and (u_2, v_2) will be removed from *path*. However, $u_0 \xrightarrow{1} f \in F$ and $v_0 \xrightarrow{1} v_0 \notin F$. So, u_0 is not equivalent to v_0 and `isEquiv(`u_0, v_0`)` will return false when the successor states of u_0 and v_0 on the symbol $a = 1$ are considered. When this returns, all pairs in path will be added to *neq* and a full iteration of the main loop will be avoided for each such pair. However, *path* only contains (u_0, v_0). `IncrementalMinimizeDFA` will still iterate over (u_1, v_1) and (u_2, v_2) even though they were already recursively passed to `isEquiv`.

Alternatively, continuing to consider Figure 2.1, note that if the loop beginning on line 7 of Algorithm 2 assigns $a = 1$ before $a = 0$, then `isEquiv(`*p,q*`)` returns false without making any recursive calls. In this case, the assertion that every pair passed recursively as an argument to `isEquiv` will not be considered later in the loop of `IncrementalMinimizeDFA` is true. In general, there does exist some $a_0 a_1 \ldots a_m \in \Sigma^*$ such that assigning $a = a_i$ first on line 7 of the ith recursive call to `isEquiv` will result in no recursive call be "wasted".

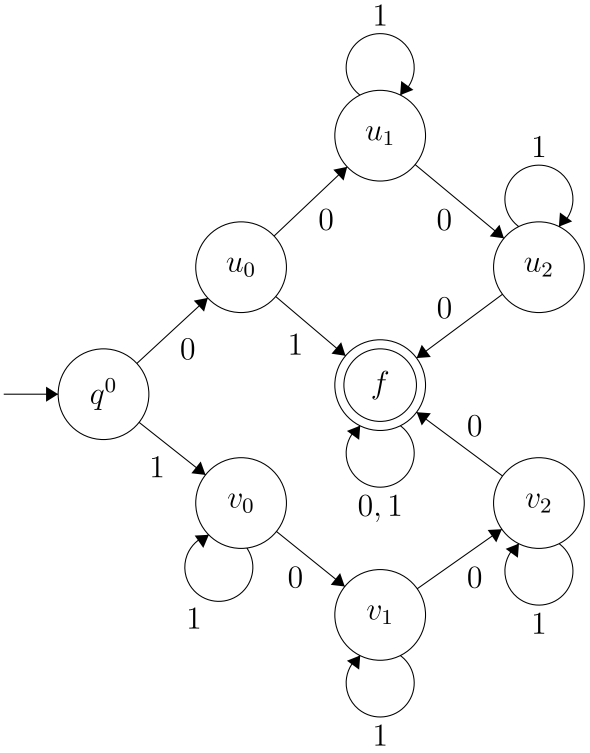

Figure 2.1: A counter-example to the claim that every recursive call to `isEquiv` prevents an iteration of the main loop in the body of `IncrementalMinimizeDFA`

However, when `isEquiv(`p,q`)` would return false, there is no effective way to predict this string. This is because any such string w would necessarily be such that the language of p and q differ on w, and whether or not the languages of p and q are equivalent is exactly what `isEquiv` is already trying to compute.

The proof given in [2] that `IncrementalMinimizeDFA` runs in worst-case $O(kn^2\alpha(n))$ time relies on the claim that every recursive call to `isEquiv` avoids an iteration of the main loop in `IncrementalMinimizeDFA`. Although this is often the case, in particular it holds whenever the initial call to `isEquiv` returns true, we have shown that it is not true generally. Revising the argument provided in [2], we have `IncrementalMinimizeDFA` has a worst case runtime of $O(kn^4\alpha(n))$. This follows from the fact that each `isEquiv` call has a runtime of $O(kn^2)$ and there is a maximum of $\frac{n^2-n}{2}$ iterations of the main loop of `IncrementalMinimizeDFA`, each of which may call `isEquiv`.

Chapter 3

Symbolic Minimization Algorithms

3.1 “Naive” Incremental Minimization

Symbolic automata are strictly more expressive than classic automata in the sense that any classic automaton can be modelled as an SFA and there exist languages (such as any over alphabets with an infinite domain) that SFAs accept but no classic automata can model. In spite of this, an arbitrary SFA $M = (\mathcal{A}, Q, q^0, F, \Delta)$ can be converted to a classic finite automaton (even if M is defined on an algebra with an infinite domain) that can be thought of intuitively as preserving the language accepted by M. As described in [8], this is done by constructing a classic automata with an alphabet defined by the set of minterms, i.e. the minimal satisfiable Booolean combinations, of predicates found in M. Specifically, for a set of predicates $P = \{\psi_1, \dots, \psi_n\}$, we define

$$Minterms(P) := \left\{ \varphi = \bigwedge_{i=0}^{n} \psi_i^* \mid \psi_i^* = \psi_i \text{ or } \psi_i^* = \neg\psi_i \text{ for } \psi_i \in P;\ \mathrm{IsSat}(\varphi) \right\}$$

We will denote the minterm set generated from the set of atomic predicates labeling the transition in M as $Minterms(M)$. Note that even if $\Psi_{\mathcal{A}}$ is infinite, there are only finitely many predicates in M by the requirement that M has finitely many transitions. Because of this, in an SFA with m transitions, $|Minterms(M)| \leq 2^m$ [8]. So, we can define a finite set $\Sigma := Minterms(M)$ as an alphabet for a classic automaton. We can define a transition set $\Delta_\Sigma \subseteq Q \times \Sigma \times Q$ set over this alphabet, and the same state set Q of M, by

$$\Delta_\Sigma := \{(p, \psi, q) \mid \psi \in \Sigma; \exists (p, \varphi, q) \in \Delta_M, IsSat(\varphi \wedge \psi)\}$$

Thus, from the SFA M we can construct a classic automaton $N = (Q, \Sigma, \Delta_\Sigma, q^0, F)$. This construction is useful because N has the following properties:

1. For all tansitions $p \xrightarrow{a} q$ in M, there is a unique corresponding transition $p \xrightarrow{\varphi} q$ in N with $a \in [\![\varphi]\!]$. In particular, if $a_1 a_2 \dots a_n \in L(M)$ then there exists a unique string $\varphi_1 \varphi_2 \dots \varphi_n \in L(N)$ such that for $1 \leq i \leq n$, $a_i \in [\![\varphi_i]\!]$.

2. For all transitions $p \xrightarrow{\varphi} q$ in N and all $a \in [\![\varphi]\!] \subseteq \mathcal{D}_{\mathcal{A}}$, there is a corresponding transition $p \xrightarrow{a} q$ in M. In particular, if $\varphi_1 \varphi_2 \dots \varphi_n \in L(N)$ then we have $a_{1j} a_{2j} \dots a_{nj} \in L(M)$ for all $a_{ij} \in [\![\varphi_i]\!]$ such that $1 \leq i \leq n$.

These properties can be utilized to adapt most algorithms defined on classic automata to symbolic automata. In particular, we have the following result:

Theorem 2. *Any algorithm for the minimization of DFAs can be adapted to an algorithm for the minimization of SFAs.*

Proof. Let $M = (\mathcal{A}, Q, q^0, F, \Delta)$ be a deterministic, clean, complete and normalized SFA and let Min_{DFA} be any algorithm that takes a DFA D as input and produces a minimal DFA D' as output with $L(D) = L(D')$. Define an algorithm Min_{SFA} by the following steps:

1. Construct a DFA $N = (Q, \Sigma, \Delta_\Sigma, q^0, F)$ with $\Sigma = Minterms(M)$ as defined above. (In particular, this gives us a DFA N such that properties 1 and 2 described above both hold)
2. Run Min_{DFA} on N to obtain a minimal DFA N'.
3. Define M' as the SFA obtained by combining the same states in M that were combined in N to produce N'.

It is claimed that M' is minimal with $L(M') = L(M)$.

To prove this, it is claimed that for states $p, q \in Q$, if $p \equiv_N q$ then $p \equiv_M q$. So, assume the statement does not hold. That is, assume there exists $p, q \in Q$ such that $p \equiv_N q$ and $p \not\equiv_M q$. So, $L_p(N) = L_q(N)$ and $L_p(M) \neq L_q(M)$. Without loss of generality, let $w = a_1 a_2 \ldots a_n \in L_p(M)$ with $w \notin L_q(M)$. By property 1 of N, there exists a string $\varphi_1, \varphi_2, \ldots, \varphi_n \in L_p(N)$ with $a_i \in [\![\varphi_i]\!]$ for $1 \leq i \leq n$. Because $L_p(N) = L_q(N)$, this implies that $\varphi_1 \varphi_2 \ldots \varphi_n \in L_q(N)$. So, by property 2 of N, we have $a_1 a_2 \ldots a_n = w \in L_q(M)$. This is a contradiction because we assumed $w \notin L_q(M)$. Thus, $p \equiv_N q$ implies $p \equiv_M q$.

By the uniqueness of minimal DFAs, $N' = N_{/\equiv_N}$ (up to the relabeling of states). So, Min_{DFA} combines states $p, q \in Q$ if and only if $p \equiv_N q$. Because $p \equiv_N q$ implies $p \equiv_M q$ and Min_{SFA} combines states in M only when the same states are combined by Min_{DFA}, $M' = M_{/\equiv_M}$. By Theorem 1, this is a minimal SFA. □

Although this adaptation process is always possible, it is generally computationally expensive. This follows from the fact that the size of $Minterms(M)$ is, in the worst case, exponential in the number of transition in M. If the runtime of the DFA minimization algorithm used has a linear dependence on the size of the alphabet, as many do, the runtime of the SFA minimization algorithm would be at least exponential in the number of states. It is because of this that algorithms generated from this process are referred to in this paper as "naive".

A naive adaptation of Hopcroft's algorithm for DFA minimization is presented in [6] (referred to there as $Min^{\mathbf{H}}_{SFA}$) with a worst-case runtime of $O(2^m f(ml) + 2^m n \log n)$ where n is the number of states in M, $m \leq n^2$ is the number of transitions, l is the size of the largest predicate, and f is the time complexity of deciding satisfiability of predicates of the given size [6]. In contrast, Hopcroft's algorithm over DFAs has a worst-case runtime of $O(kn \log n)$ where k is the alphabet size [10].

In practice, as demonstrated by the naive adaptation of Hopcroft's algorithm in [6], a DFA need not actually be constructed to apply this algorithm. Rather, we can simply treat the input SFA as if it was a DFA with an alphabet defined by its minterms. This is how we'll first approach adapting `IncrementalMinimizeDFA` as given in Algorithm 1 to symbolic automata. Algorithm 3 is the result of this adaptation.

The body of Algorithm 1 is largely unchanged when adapting to symbolic automata. This is because we only make assumptions about the automaton's alphabet in the call to `isEquiv`. Aside from the input, the only change made was to initialize *neq* to contain all pairs of states (p, q) such that $\texttt{Dist}(p) \neq \texttt{Dist}(q)$. Unlike in Algorithm 3, this requires some precomputation: a breadth first search of the automaton is required to label each state by its distance to an accepting state. This would take $O(m)$ time, where $m \leq n^2$ is the number of transitions in M. This precomputation is not strictly necessary, *neq* may still be initialized as in Algorithm 1. However, repeated recursive calls to `isEquiv` are considerably expensive. So, it is beneficial to be able to identify a witness to non-equivalence in as few calls as possible.

```
Function IncrementalMinimize($M = (Q, A, q^0, \Delta, F)$):
    for $q \in Q$ do
        Make($q$)
    $neq = \{\texttt{Normalize}(p, q) \mid p \in Q, q \in Q, \texttt{Dist}(p) \neq \texttt{Dist}(q)\}$
    for $p \in Q$ do
        for $q \in \{x \in Q \mid x > p\}$ do
            if $(p, q) \in neq$ then
                continue
            if Find($p$) = Find($q$) then
                continue
            $equiv, path = \varnothing$
            if isEquiv $(p,q)$ then
                for $((p', q') \in equiv)$ do
                    Union($p'$,$q'$)
            else
                for $(p', q') \in path$ do
                    $neq = neq \cup \{(p', q')\}$
    return JoinStates($M$)
```

Algorithm 3: Extension of algorithm 1 to symbolic automata

The body of `isEquiv` for the symbolic algorithm is given in Algorithm 4. For the naive adaptation of `isEquiv`, only a few changes need be made to adapt the DFA case to all symbolic automata. Specifically, we replace the iteration over Σ on line 7 with an iteration over $Minterms(M)$. The construction of this minterm set also requires precomputation and the process is outlined in Figure 2 of [6]. This construction starts with a binary tree with root $\top$. In brief, if M contains atomic predicates $\varphi_1, \ldots, \varphi_m$, then this tree is built by setting φ_i and $\neg\varphi_i$ as children to all nodes on level i. After this construction is terminated, the minterms are the conjunction of the predicates along each path from the root to a leaf (as long as this conjunction is satisfiable).

```
1  Function isEquiv($p, q$):
2      if $(p, q) \in neq$ then
3          return False
4      if $(p, q) \in path$ then
5          return True
6      $path = path \cup \{(p, q)\}$
7      for $\varphi \in Minterms(M)$ do
8          $(p', q')$ = Normalize(Find($\delta(p, \varphi)$), Find($\delta(q, \varphi)$))
9          if $p' \neq q'$ and $(p', q') \notin equiv$ then
10             $equiv = equiv \cup \{(p', q')\}$
11             if not isEquiv($p', q'$) then
12                 return False
13     $path = path \setminus \{(p', q')\}$
14     $equiv = equiv \cup \{(p, q)\}$
15     return True
```

Algorithm 4: ”Naive” equivalence check for symbolic automata

Because a binary tree with height m will have at most 2^m leaves, it is immediately evident that we may have up to 2^m minterms for an SFA with m transitions. It follows that the minterm computation can have a worst case runtime of $O(2^m f(ml))$ where l is the length of the largest predicate in M and f is the complexity of deciding predicate satisfiability [6]. Additionally because $|Minterms(M)| \leq 2^m$, the loop starting on line 7 of Algorithm 4 will not terminate for 2^m iterations in the worst case. This is why repeated recursive calls to `isEquiv` becomes so expensive in the symbolic setting. Overall, the runtime of Algorithm 3 is $O(2^m f(ml) + 2^m n^4 \alpha(n))$ where $n = |Q|$.

3.2 Efficient Incremental Minimization

Intuitively, we iterate over $Minterms(M)$ in line 7 of Algorithm 4 because $Minterms(M)$ completely partitions the domain $\mathcal{D}$ of the alphabet $\mathcal{A}$ of the SFA M. The predicate set Ψ of $\mathcal{A}$ need not define a partition on the domain. For instance, we could have a case where $\mathcal{D} = \mathbb{Z}$ and Ψ contains atomic predicates $\varphi_{\leq 0}$ and $\varphi_{\geq 0}$ where $[\![\varphi_{\leq 0}]\!] = \{n \in \mathbb{Z} \mid n \leq 0\}$ and $[\![\varphi_{\geq 0}]\!] = \{n \in \mathbb{Z} \mid n \geq 0\}$. In this case, our algorithm needs to be able to recognize that transition $p \overset{\varphi_{\leq 0}}{\rightarrow} p'$ and $q \overset{\varphi_{\geq 0}}{\rightarrow} q'$ overlap on the input character $0 \in \mathcal{D}$ but are disjoint on all other domain characters. Algorithm 4 resolves this by partitioning the domain over the minterm set such that each domain character is contained in the denotation of at most one minterm. However, in general, we need not partition the domain so finely.

Instead of partitioning on $Minterms(M)$, we can take advantage of the fact that `isEquiv` only needs to partition the domain over the outgoing transitions of a single pair of states at any one time. If (p, q) is this pair then we can replace the iteration over $Mitnterms(M)$ in line 7 of Algorithm 4 with any set $\Phi_{p,q} \subseteq \Psi$ such that for all $\varphi \in \Phi_{p,q}$, there exists at most one transition $p \xrightarrow{\psi_1} p'$ and at most one transition $q \xrightarrow{\psi_2} q'$ such that for all $a \in [\![\varphi]\!]$, $a \in [\![\psi_1]\!]$ and $a \in [\![\psi_2]\!]$. For instance, we could define $\Phi_{p,q} = Minterms(P_{p,q})$ where $P_{p,q} \subseteq \Psi$ is the set of all predicates labeling transitions coming out of p or q in M. $Minterms(P_{p,q})$ will be referred to as the local minterm set for pair (p, q). Because p and q each have at most $n = |Q_M|$ outgoing transitions, iterating over this set in `isEquiv` would reduce the worst case number of iterations from $2^m \leq 2^{n^2}$ in the worst case to 4^n.

So, one possible approach to optimizing `isEquiv` as given in Algorithm 4 would be to compute the set $Minterms(P_{p,q})$ and then iterate over this set instead of $Minterms(M)$. However, generating this minterm set for each call to `isEquiv` would be very computationally expensive. Moreover, much of this computation would often be wasted. One can imagine a scenario where after computing the local minterm set, the first recursive call to `isEquiv` immediately returns false.

```
Function isEquiv($p, q$):
    if $(p, q) \in neq$ then
        return False
    if $(p, q) \in path$ then
        return True
    $path = path \cup \{(p, q)\}$
    $Out_p = \{\varphi \in \Psi_A \mid \exists p', (p, \varphi, p') \in \Delta\}$ // All outgoing predicates of $p$
    $Out_q = \{\psi \in \Psi_A \mid \exists q', (q, \psi, q') \in \Delta\}$ // All outgoing predicates of $q$
    while $Out_p \cup Out_q \neq \varnothing$ do
        Let $a \in [\![(\bigvee_{\varphi \in Out_p} \varphi) \wedge (\bigvee_{\psi \in Out_q} \psi)]\!]$
        $(p', q') =$ Normalize(Find($\delta(p, a)$), Find($\delta(q, a)$))
        if $p' \neq q'$ and $(p', q') \notin equiv$ then
            $equiv = equiv \cup \{(p', q')\}$
            if not isEquiv($p', q'$) then
                return False
        Let $\varphi \in Out_p$ with $a \in [\![\varphi]\!]$
        Let $\psi \in Out_q$ with $a \in [\![\psi]\!]$
        if $IsSat(\varphi \wedge \neg\psi)$ then
            $Out_p = Out_p \setminus \{\varphi\} \cup \{\varphi \wedge \neg\psi\}$
        if $IsSat(\psi \wedge \neg\varphi)$ then
            $Out_q = Out_q \setminus \{\psi\} \cup \{\psi \wedge \neg\varphi\}$
    $path = path \setminus \{(p, q)\}$
    $equiv = equiv \cup \{(p, q)\}$
    return True
```

Algorithm 5: A more efficient equivalence check for states p and q

A more efficient version of `isEquiv` is presented in Algorithm 5. Intuitively, it can be thought of as iterating through such a local minterm set without needing to construct it upfront. Instead, we initialize sets Out_p and Out_q to contain the predicates labeling the outgoing transitions of p and q, respectively. During each iteration of the loop in `isEquiv`, we pick a character a that satisfies a predicate $\varphi \in Out_p$ and Out_q, test the next pair of states after transitioning p and q on a, and then remove the local minterm a satisfies from Out_p and Out_q.

The body of `IncrementalMinimize` need not be changed from Algorithm 3. A full proof of correctness for `IncrementalMinimize` with `isEquiv`, as defined in Algorithm 5, is included below.

Lemma 3. `isEquiv` *terminates.*

Proof. First, note that there are a finite number of recursive calls to `isEquiv`. This is because there are a finite number of pairs of states that `isEquiv` can be called on and `isEquiv` immediately returns if it recognizes that it has already been called on a given pair of states. So, to prove `isEquiv` terminates, it needs only be shown that the loop over $Out_p \cup Out_q$ begininning on line 9 is finite.

During each iteration over $Out_p \cup Out_q$, we find some $\varphi \in Out_p$ and $\psi \in Out_q$ such that there exists an $a \in \mathcal{D}_{\mathcal{A}}$ with $a \in [\![\varphi \wedge \psi]\!]$. Later, during the same iteration, we replace φ in Out_p with $\varphi \wedge \neg\psi$ and ψ in Out_q with $\psi \wedge \neg\varphi$ (as long as these predicates are satisfiable). These new predicates denote strictly smaller subsets of $\mathcal{D}_{\mathcal{A}}$ (because a does not satisfy either predicate). If $\mathcal{D}_{\mathcal{A}}$ is finite, this is enough to ensure the loop terminates. Otherwise, if $\mathcal{D}_{\mathcal{A}}$ is infinite, it needs to be proven that for all $\varphi \in Out_p$, there exist some finite set $S \subseteq Out_q$ such that $[\![\varphi \wedge \neg(\bigvee_{\psi \in S} \psi)]\!] = \varnothing$. Because Out_p is always finite, this is sufficient to prove that the loop terminates.

Fix $\varphi \in Out_p$. Define $S = \{\psi \in Out_q \mid IsSat(\varphi \wedge \psi)\}$. Assume that $\varphi \wedge \neg(\bigvee_{\psi \in S} \psi)$ is satisfiable. Therefore, there exists some $a \in [\![\varphi]\!]$ such that $a \notin \bigvee_{\psi \in S} \psi$. We will inductively prove that this is a contradiction on every iteration of this loop such that $\varphi \in Out_p$.

During the first loop iteration, Out_q is equivalent to the predicates of the outgoing transitions of q. Because M is compltete, there exists some $\psi \in Out_q$ such that $a \in [\![\psi]\!]$. So, $\varphi \wedge \psi$ is satisfiable and ψ is an element of S which is a contradiction because $a \notin [\![\bigvee_{\psi \in S} \psi]\!]$. Beyond the first iteration, assume that there exists some $\psi \in Out_q$ at the start of the iteration such that $a \in [\![\psi]\!]$. There are two cases which we must consider:

1. If ψ is not removed from Out_q during this iteration of the loop, then $\psi \wedge \varphi$ is satisfiable and $a \in [\![\psi]\!]$ which is a contradiction.

2. If ψ is removed from Out_q during this iteration, then there exists some $\varphi' \neq \varphi$ in Out_p with $IsSat(\psi \wedge \varphi')$. At the end of this iteration, $\psi \wedge \neg\varphi'$ replaces ψ in Out_q. However, because M is deterministic and $\varphi \neq \varphi'$, $a \notin [\![\varphi']\!]$. Therefore, $a \in [\![\psi \setminus \varphi']\!]$ and $\varphi \wedge (\psi \wedge \varphi')$ is satisfiable. Because $\psi \wedge \varphi' \in Out_q$, this is a contradiction.

So, for any given point in the iteration of this loop, $\varphi \in Out_p$ implies that there exists a finite set $S \subseteq Out_q$ such that $[\![\varphi \wedge \neg(\bigvee_{\psi \in S} \psi)]\!] = \varnothing$. This ensures that the loop over $Out_p \cup Out_q$ is finite. Therefore, `isEquiv` terminates. □

Lemma 4. *A call to* `isEquiv` *from the body of* `IncrementalMinimize` *returns true if and only if the states p and q of SFA M initially passed to it are equivalent.*

Proof. `isEquiv` returns false on (p, q) only if the pair (p, q) is contained in *neq*, which only contains pairs of states known to be distinguishable, or if a recursive call to `isEquiv` returns false. In the later case, we know that p, q must not be equivalent because we have found a string $w \in \mathcal{D}_{\mathcal{A}}^*$ such that $\delta(p, w)$ and $\delta(q, w)$ are known to be distinguishable.

A recursive call to `isEquiv` returns true only if (p, q) is contained in *path*, which only occurs if a cycle of indistinguishable states is found, or if all of its recursive calls to `isEquiv` return true. Therefore, when called from the body of `IncrementalMinimize`, `isEquiv` returns true only if for all $w \in \mathcal{D}_{\mathcal{A}}^*$, $\delta(p, w)$ is either known to be equivalent or is indistinguishable from $\delta(p, w)$. Therefore, p, q are equivalent. □

Lemma 5. *If a call to* `isEquiv` *from the body of* `IncrementalMinimize` *returns true, equiv contains only pairs of states (p, q) such that p and q are equivalent. If a call to* `isEquiv` *from the body of* `IncrementalMinimize` *returns false, then path contains only pairs of states (p, q) such that p and q are distinguishable.*

Proof. *equiv* is a set of pairs of states such that for all $(p', q') \in equiv$ there exists some $w \in \mathcal{D}_{\mathcal{A}}^*$ with $p' = \delta(p, w)$ and $q' = \delta(q, w)$. If `isEquiv` returns true on (p, q) then p and q are equivalent by the previous lemma. So, for all $w \in \mathcal{D}_{\mathcal{A}}^*$, $\delta(p, w)$ is equivalent to $\delta(q, w)$. Therefore, each pair of states in *equiv* contains equivalent states.

path is a set of pairs of states that initially contains (p_0, q_0), the initial arguments passed to `isEquiv`. From that it tracks the path of `isEquiv` in the depth first traversal of the automata's set of states. That is, for all $(p_i, q_i) \in path$, either $i = 0$ or there exists $(p_{i-1}, q_{i-1}) \in path$ with $p_i = \delta(p_{i-1}, a)$ and $q_i = \delta(q_{i-1}, a)$ for some $a \in \mathcal{D}_{\mathcal{A}}$. If `isEquiv` returns false on (p_0, q_0), then, every recursive call to $(p_i, q_i) \in path$ returns false. Since the contents of *path* are not changed if `isEquiv` returns false, every pair of states in *path* is distinguishable. Hence, these are added to *neq*. □

Theorem 6. *Running* `IncrementalMinimize` *on M until termination returns an SFA M' such that M' is minimal and $L(M) = L(M')$.*

Proof. Consider the loop starting in line 5 in `IncrementalMinimize`. This loop checks for all normalized pairs of states p, q for equivalence (if the states have same minimum distance to accepting set of states). Each equivalence check is performed by a call to `isEquiv`. If `isEquiv` returns true then, from Lemma 5, every pair of states in *equiv* (including the initial arguments) are equivalent.

Hence, when the loop terminates, the equivalence check on all pairs of states is performed and all possible pairs of equivalent states would be identified. Since `IncrementalMinimize` only merges states that are proved to be equivalent (line 14), all equivalent states would be merged into the same equivalence class. Since all the states that are not merged are not-equivalent, `IncrementalMinimize` returns the minimal symbolic automata. □

Note that the proof of Lemmas 4 and 5 as well as the proof of Theorem 6 are closely adapted from the proofs of the DFA case in [2].

Theorem 7. `IncrementalMinimize` *has a worst case runtime of $O(n^6 f(2nl)\alpha(n))$ where n is the number of states in M, l is the length of the longest predicate labeling a transition in M and f is the time complexity of determining predicate satisfiability of a given length.*

Proof. First, note that `isEquiv` runs in $O(n^2kf(2nl))$ time where k is the number of iterations over $Out_p \cup Out_q$ beginning on line 9. This follows from the fact that `isEquiv` makes at most n^2 recursive calls (one for every possible pair of states, in the worst case) and the satisfiability check on line 10 of `isEquiv` takes $f(2nl)$ time.

Each iteration over $Out_p \cup Out_q$ selects some $\varphi \in Out_p$ and $\psi \in Out_q$ such that there exists an $a \in \mathcal{D}_{\mathcal{A}}$ with $a \in [\![\varphi \wedge \psi]\!]$.During the same iteration, φ is removed from Out_p and replaced $\varphi \wedge \neg\psi$ (if this predicate is satisfiable). By the proof to Lemma 3, this new predicate must be unstatisfiable after $|S|$ many iterations (possibly non-consecutive) iterations where $S = \{\psi \in Out_q \mid IsSat(\varphi \wedge \psi)\}$. Because $S \subseteq Out_q$ and Out_q is bounded by n, the maximum number of outgoing transitions from q, $|S| \leq n$. Because the size Out_p is also bounded by n, it follows that it takes n^2 many iterations before $Out_p = \varnothing$.

When $Out_p = \varnothing$, we also must have $Out_q = \varnothing$. To prove this, assume that after some iteration of the loop beginning on line 9 of `isEquiv`, we have $Out_p = \varnothing$ but $Out_q \neq \varnothing$. Then, there exists some satisfiable $\psi \in Out_q$. Fix $a \in [\![\psi]\!]$. Because M is complete, there is a transition $p \xrightarrow{\varphi} p'$ in M with $a \in [\![\varphi]\!]$. So, $\varphi \in Out_p$ when Out_p is initialized on line 7. Because we eventually reach the case where $Out_p = \varnothing$, there existed some iteration where $\varphi_i \in Out_p$ with $a \in [\![\varphi_i]\!]$ is replaced in Out_p with $\varphi_{i+1} := \varphi_i \wedge \neg\psi_i$ (if this predicate is satisfiable) with $a \notin [\![\varphi_{i+1}]\!]$. So, $a \in [\![\psi_i]\!]$. Therefore, $IsSat(\psi_i \wedge \psi)$. Because M is deterministic and normalized, this implies $\psi_i = \psi$. However, in the same iteration that φ_i is removed from Out_p, $\psi_i = \psi$ is removed from Out_q. This contradicts the assumption that after some iteration of the loop, $Out_p = \varnothing$ and $\psi \in Out_q$. Therefore, the loop terminates when $Out_p = \varnothing$, after n^2 iterations.

So, $k = n^2$ and the worst-case runtime of `isEquiv` is $O(n^4f(2nl)\alpha(n))$. Following the proof of the runtime of `IncrementalMinimizeDFA` given in Section 2.3 (revised from [2]), we have $\frac{n^2-n}{2}$ iterations of the main loop of `IncrementalMinimize` each of which may call `isEquiv` once. So, the worst case runtime of `IncrementalMinimize` for SFAs is $O(n^6f(2nl)\alpha(n))$ ☐

In contrast, the efficient adaptation of Hopcroft's algorithm to symbolic automata in [6] runs in $O(n^2 \log(n) f(nl))$ time.

3.3 Minimization with Dependency Checking

There is one downside to all implementations of `isEquiv` given thus far. That is, we only ever merge equivalence classes in the case where all recursive calls to `isEquiv` return True. In Section 2.3, we gave an example of a DFA where `isEquiv` is called on a pair of states (u_0, v_0) and a recursive call to `isEquiv` establishes the indistinguishability of a pair of states (u_1, v_1). However, because a separate recursive call to `isEquiv` returns false, the equivalence classes of u_1 and v_1 are not merged and the computation is wasted. So, we seek to avoid this waste.

This is not so simple though. `isEquiv` works by assuming that the tested pair is equivalent for later recursive calls unless non-equivalence can immediately be proven. So, except for the initial call from the body of `IncrementalMinimize`, `isEquiv` returning true need not imply that the test pair is distinguishable. Rather, a recursive call returning true only implies that we have not yet found a witness to non-equivalence. To help identify when a call on `isEquiv` returning true actually indicates that a given pair is equivalent, as we traverse the `isEquiv` recursive call tree, we can track the pair's "dependencies".

That is, our algorithm is already based on the idea that state p is equivalent to state q if for all states p' and q' such that $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ for some $a \in \mathcal{D}$, we have p' is equivalent to q'. If `isEquiv`(p, q) is called and can not immediately prove non-equivalence, by finding $(p, q) \in neq$, then it is assumed to be equivalent and `isEquiv` is called on the pair of successor states for each character in the domain. If we have a data structure that tracks these assumptions then we can identify whether or not we've found any states to be equivalent even if `isEquiv` returns false. All such assumptions made while executing `isEquiv`(p, q) will be called the dependencies of p, q. A formal definition is provided below.

Definition 6. A pair (p, q) is *dependent* on $(\hat{p}, \hat{q})$ (denoted $(\hat{p}, \hat{q}) \in$ `Dependencies`(p,q)) if and only if one of the following holds

1. (p', q') is assigned the value $(\hat{p}, \hat{q})$ on line 11 of `isEquiv` as given in Algorithm 5.

2. There exists a pair of states (r, s) such that $(r, s) \in$ `Dependencies`(p,q) and $(\hat{p}, \hat{q}) \in$ `Dependencies`(r,s) (i.e. the dependency relation is transitive)

In general, we have p is equivalent to q if and only if $\hat{p}$ is equivalent to $\hat{q}$ for all $(\hat{p}, \hat{q}) \in$ `Dependencies`(p,q). There is a relatively simple way to check if this condition is satisfied within the context of our algorithm.

Theorem 8. *If* `isEquiv`(p, q) *is a recursive call and returns true, then* $L_p(M) = L_q(M)$ *if and only if* $(\hat{p}, \hat{q}) \notin path_{final}$ *for all* $(\hat{p}, \hat{q}) \in$ `Dependencies`(p,q)*, where* $path_{final}$ *is the value of path after the initial call to* `isEquiv` *from the body of* `IncrementalMinimize` *terminates.*

Proof. The forward direction of this implication is immediately true. If we have $(\hat{p}, \hat{q}) \in$ `Dependencies`(p,q), then there exists some $w \in \mathcal{D}^*$ such that $p \xrightarrow{w} \hat{p}$ and $q \xrightarrow{w} \hat{q}$. So, if $L_p(M) = L_q(M)$, then $\hat{p}$ must be equivalent to $\hat{q}$.

For the other direction of the implication, first note that if $path_{final}$ is empty, then we have one of the following cases:

1. The initial call to `isEquiv` returned false without any recursive calls. Because no recursive calls were made to `isEquiv`, the theorem is vacuously true.

2. The initial call to `isEquiv` returned true. By Lemma 5, all pairs of states stored in *equiv* are equivalent. By lines 13 of Algorithm 5, if `isEquiv` has been recursively called on (p, q) then $(p, q) \in equiv$. Thus, p is equivalent to q and the theorem holds.

So, assume that $path_{final} \neq \varnothing$. Therefore, the initial call to `isEquiv` returns false.

Let (p, q) be a pair of states such that `isEquiv`(p,q) was recursively called and returned true and such that$(\hat{p}, \hat{q}) \notin path_{final}$ for any $(\hat{p}, \hat{q}) \in$ `Dependencies`(p,q). Without loss of generality, let (p, q) be the first such pair of states that `isEquiv` is recursively called on (so, `isEquiv`(p,q) is not returning true because $(p, q) \in path$). Proceeding by contradiction, assume that $L_p(M) \neq L_q(M)$. Because `isEquiv`(p,q) returns true, there must be some pair of states $(r, s) \in$ `Dependencies`(p,q) that are assumed to be equivalent (i.e. $(r, s) \in path$ or

$(r, s) \in equiv$) but are actually not equivalent. Because `isEquiv(`p,q`)` returns false when not called recursively (by Lemma 4), we must have that (r, s) was added to *equiv* or *path* outside of the context of the `isEquiv(`p,q`)` call. So, `isEquiv(`r,s`)` was recursively called before `isEquiv(`p,q`)`. Because $(r, s) \in$ `Dependencies(`p,q`)`, we have $(r, s) \notin path_{final}$. Therefore, because (r, s) is added to *equiv* or *path*, `isEquiv(`r,s`)` does not return false. In particular, it must return true. However, `isEquiv(`r,s`)` is called before `isEquiv(`p,q`)` and it was assumed that `isEquiv(`p,q`)` was the earliest recursive call to return true such that $(\hat{p}, \hat{q}) \notin path_{final}$ for all $(\hat{p}, \hat{q}) \in$ `Dependencies(`p,q`)`. Therefore, there must exist some pair of states $(\hat{r}, \hat{s}) \in$ `Dependencies(`r,s`)` such that $(\hat{r}, \hat{s}) \in path_{final}$. However, dependencies are defined to be transitive. That is, $(r, s) \in$ `Dependencies(`p,q`)` and $(\hat{r}, \hat{s}) \in$ `Dependencies(`p,q`)` implies $(\hat{r}, \hat{s}) \in$ `Dependencies(`p,q`)`. So, we must have $(\hat{r}, \hat{s}) \notin path_{final}$. This is a contradiction. Therefore, we have $L_p(M) = L_q(M)$. □

Thus, we have found a sufficient condition for identifying a pair of states as equivalent in the context of a recursive `isEquiv` call that returns true. We can now modify `isEquiv` as given in Algorithm 5 to check for this condition whenever it returns false and merge states accordingly. Details about how this check can be implemented using a directed graph to represent the dependency relation will be given in Section 4.2. Unfortunately, as discussed in Chapter 5, the dependency checking algorithm that was implemented for evaluation does not perform better in practice than the efficient incremental algorithm given in Section 3.2.

Chapter 4

Implementation

The algorithms presented in this paper were implemented[1] in Java using the *Symbolic Automata* library[2]. For purposes of optimization, there are a few key differences between the pseudoceode algorithms presented previously and their actual implementation.

4.1 Incremental Minimization

The implementation of `IncrementalMinimize` is very similar to the pseudocode presented in Algorithm 3. One might reasonably expect that a heuristic could be identified for the order in which states are iterated on lines 5 and 6. One such possibility is to order states based on minimum distance to an accepting state so that states closest to an accepting state are tested first. The heuristic would hopefully result in quicker calls to `isEquiv` as fewer recursive calls would ideally need to be made. Additionally, as we are already computing minimum distance to an accepting state for the initialization of *neq*, one might hope that no further computation needs to be performed to determine this ordering. Unfortunately, this is not quite the case.

The initialization of *neq* proceeds by a traversal of the graph that the automata represents. During this traversal, accepting states are associated to the integer zero in a hash map. The predecessors of accepting states are associated to the integer one (unless already in the hash map), and their predecessors with the integer two, and so on. When this process terminates, the hash map contains a nearly complete mapping of states to their minimum distance to an accepting state. However, states that can not reach an accepting state are excluded from this map. Our implementation solves this by wrapping *neq* inclusion queries in such a way that test for inclusion in the *neq* set first, then tests if either states are included in the hash map. If both states are included within the hash map and they are mapped to different values, then they are necessarily non-equivalent. Alternatively, if one of the states is included within the hash map while the other is not, then the states are also necessarily inequivalent. This wrapper implementation prevents us from needing to extract all pairs of states with varying minimum distances from the hash map to add to *neq*, as Algorithm 3 indicates. Additionally, it prevents us from needing to perform the $O(n)$ operation necessary to iterate through all

---

[1] https://github.uconn.edu/jah12014/symbolic-automata-research

[2] https://github.com/lorisdanto/symbolicautomata [5]

of the states to identify which are not included in the hash map and then map those states to infinity.

However, this wrapper solution does not lend itself to iteration. We could replace the hash map with a linked hash map to preserve the order that keys are placed in the map, and then iterate over that. However, this approach would still run into the issue with states that can't reach an accepting state being ignored. So, in order to implement the desired heuristic for iteration order, the additional $O(n)$ precomputation time would still be required. In practice, iterating through states in the order of their minimum distance to an accepting state did not noticeably improve on the performance the algorithm.

Unlike `IncrementalMinimize`, the efficient `isEquiv` algorithm, as given in Algorithm 5, was changed significantly in the process of implementation. Specifically, because recursion is generally inefficient in Java and has the risk of stack overflow errors for very large call trees, it was decided to implement `isEquiv` without recursion. A class `EquivRecord` is defined to hold the parameters p and q that would have been recursively passed to `isEquiv`. The value of *path* is also stored in `EquivRecord` because it is no longer usable as a global variable consistent across recursive calls. When `isEquiv` is initially called, it creates an `EquivRecord` instance from the states initially passed and adds it to a stack. The main body of `isEquiv` is wrapped in a loop that continues until the stack is empty and instead of recursively calling `isEquiv` on pairs of successor states, a new instance of `EquivRecord` is created and pushed onto the stack. The full implementation of `isEquiv` without recursion can be seen in Appendix A. In practice, the non-recursive implementation of `isEquiv` tended to significantly outperform an implementation with recursion.

4.2 Dependency Checking

Because the dependency relation described in Section 3.3 is transitive, it is natural to represent these dependencies as a directed graph. The vertices of this graph are pairs of states and a directed path existing from (p, q) to (p', q') indicates that (p, q) is dependent on (p', q'). We can construct such a graph as we iterate through pairs of states in a call to `isEquiv`: add a vertex for the initial pair (p, q) and whenever a new (p', q') is assigned on line 11 of Algorithm 5, we add a vertex for (p', q') with a directed edge from (p, q) to (p', q') unless `Find(`p`)` = `Find(`q`)` (i.e. unless we already know that p is equivalent to q). In this construction, `Dependencies(`p,q`)` is the set of all pairs reachable within this graph from the vertex containing (p, q).

Unfortunately, our non-recursive implementation of `isEquiv` does not lend itself well to dependency checking. This is because our proof of Theorem 8 relies on the recursive nature of the algorithm. Without recursion, it is difficult to identify when a call to `isEquiv` would have returned true and thus, it is difficult to identify when states can be merged. One solution to this, is to add a boolean value to each vertex of the graph indicating whether the pair has been tested in the loop of `isEquiv`. Then, when `isEquiv` returns false, we can merge states if all of their successors in the dependency graph have been tested and none of the successors are included in the *path* that `isEquiv` returns false on.

Additionally, a recursive implementation of dependency checking lends itself well to optimization, while the non-recursive implementation does not. Specifically, when `isEquiv` is implemented recursively, we need not add a dependency to the graph for all successor pairs. Rather, only whenever `isEquiv` assumes a pair of states to be true because they already exist within *equiv* or *path*. This is because, within the context of a recursive call that returns true, the only assumptions of equivalence that need be verified are those initially made outside that context. However, this is not replicable in a non-recursive implementation and so, we are required to add a dependency to the graph for all pairs of successor states not known already to be equivalent. This significantly increases the size of the graph the we are required to traverse to identify pairs of states that can be merged.

Because of these issues, the implementation of dependency checking is generally inefficient when compared to the efficient incremental algorithm without dependency checking. This will be explored further in the next chapter.

Chapter 5

Experimental Evaluation

The performance of our algorithms has been evaluated on a set of 2115 symbolic automata with under 400 states generated by parsing regular expressions. These regular expressions were acquired from [5] and initially sourced from the regular expressions library RegExLib[1]. All results presented within this section are the product of running minimization algorithms on this test set and averaged across five trial runs.

5.1 Comparison of Incremental Algorithms

First, we'll compare the implementations of the incremental algorithms introduced in this thesis. Figure 5.1 compares the runtime of the naive incremental algorithm, based on minterm generation as described in Section 3.1, to the efficient incremental algorithm described in Section 3.2. The runtime of the efficient incremental algorithm is consistently better than the naive algorithm across automata of all state space sizes. In fact, the efficient incremental algorithm performed better than the naive algorithm on all automata tested with five or more states. As can be seen in Figure 5.1, the performance of the naive algorithm is also more prone to spiking. This is because of the algorithm's dependence on the number of distinct predicates in an automaton.

Figure 5.2 displays the performance of the efficient incremental algorithm with and without dependency checking. Of the 2114 automata tested, the dependency checking algorithm performed better on average in 851 cases. However, in all of these cases, the dependency checking algorithm never performed better by more that 0.4 ms. On the other hand, the algorithm was 30.4 seconds slower in the worst case. The figure indicates that although we are able to identify situations were computation is wasted with dependency checking and merge states that are found to be equivalent, the computation required for this identification is not worth it. Additionally, the performance of the dependency checking algorithm tends to spike even more than the naive algorithm. This is likely because of the computational overhead needed to check whether any state can be merged whenever `isEquiv` returns false, even when no states can be merged. In these cases, our attempt to prevent wasting computation simply exasperates the issue of wasted computation. Still, the dependency checking algorithm does perform better in 851 cases, even if by a barely noticeable amount. It is

---

[1] http://www.regexlib.com

because of this that the dependency checking algorithm is believed to be worth analyzing and also worth future optimization efforts.

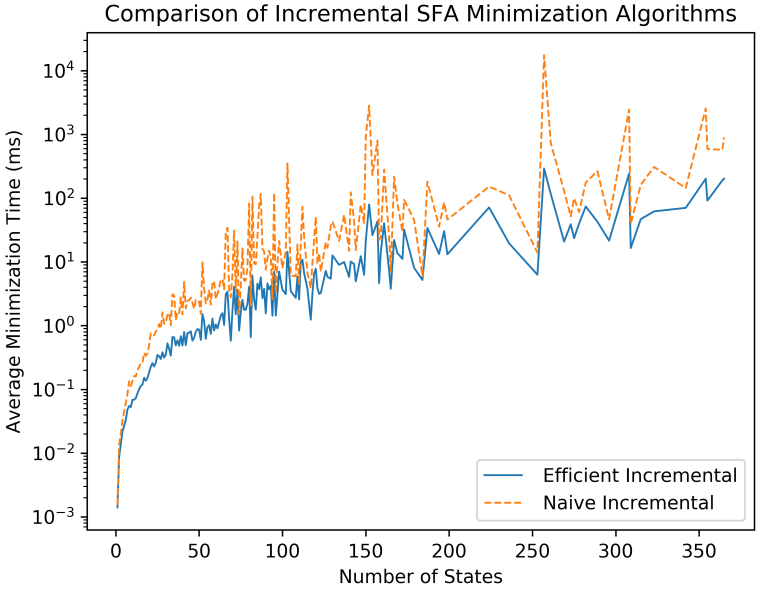

Figure 5.1: Average runtimes of the naive incremental algorithm with minterm generation and the efficient incremental algorithm without minterm generation

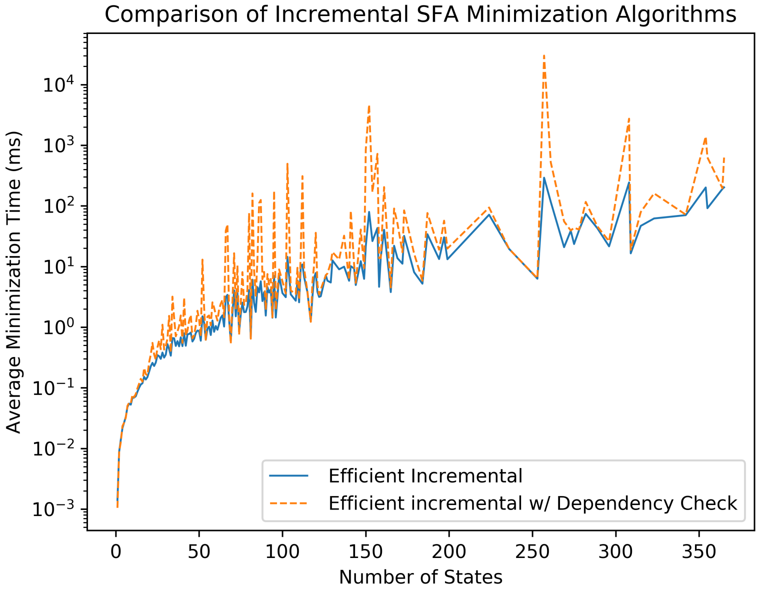

Figure 5.2: Average runtimes of the efficient incremental algorithm and incremental algorithm with dependency checking

5.2 Comparison with Pre-existing SFA Minimization Algorithms

Extensions of both Moore's algorithm and Hopcroft's algorithm for DFA minimization to symbolic automata are presented in [6]. In particular, [6] gives a "naive" adaptation of Hopcroft's algorithm, referred to as $Min^{\mathbf{H}}_{SFA}$, using minterm generation as described in Section 3.1, and an efficient modification of Hopcroft's algorithm, referred to as $Min^{\mathbf{N}}_{SFA}$, that does not require minterm generation. This efficient extension of Hopcroft's algorithm has the best asymptotic performance of all minimization algorithms for deterministic symbolic automata known to the author of this thesis. Figure 5.3 compares the efficient incremental minimization algorithm presented in Section 3.2 to Moore's algorithm and the efficient variation of Hopcroft's algorithm, both implemented as described in [6]. Hopcroft's algorithm with minterm generation was not implemented for comparison because it is consistently outperformed in all contexts by the modification of Hopcroft's algorithm without minterm generation.

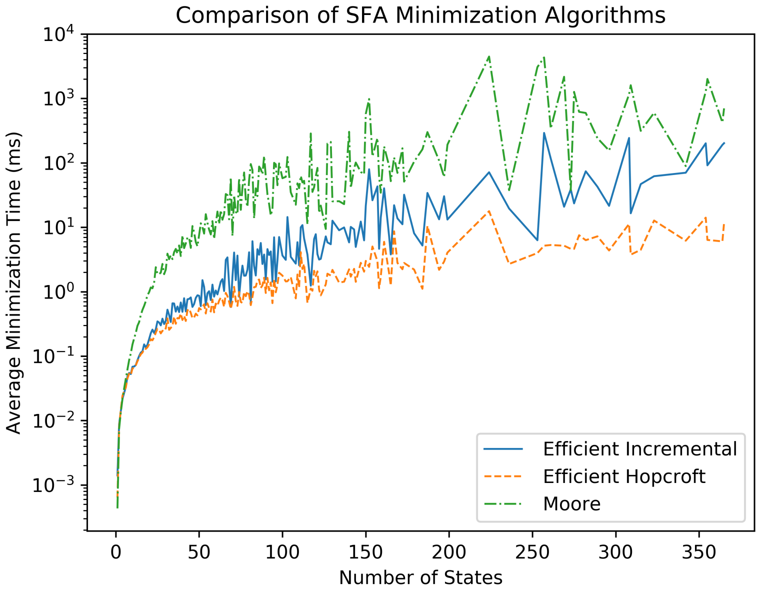

Figure 5.3: Comparison of average runtime of the efficient incremental algorithm to the minimization algorithms given in [6]

For all automata tested, the incremental algorithm tends to outperform Moore's algorithm for SFAs. Additionally, for automata of small size (i.e. under 50 states), the performance of the incremental algorithm is generally comparable to the performance of the efficient modification of Hopcroft's algorithm. However, the modified Hopcroft algorithm does typically have the best average runtime. The incremental algorithm. The performance of these algorithms tend to be closest when the given automata is already minimized or near-minimized.

5.3 Evaluation of Incremental Performance

None of our incremental algorithms improve on the average runtime of pre-existing SFA minimization algorithms. However, this drop in performance is a deliberate trade-off for the incremental behavior of the algorithms. Specifically, the incremental algorithms can be halted at any time to return a partially minimized SFA that accepts the same language as the input automaton. In contrast, the execution of the algorithms presented in [6] must finish before any automaton, aside from the one given as input, can be returned. In this section, we'll analyze the incremental progress of minimization that occurs before the algorithms terminate.

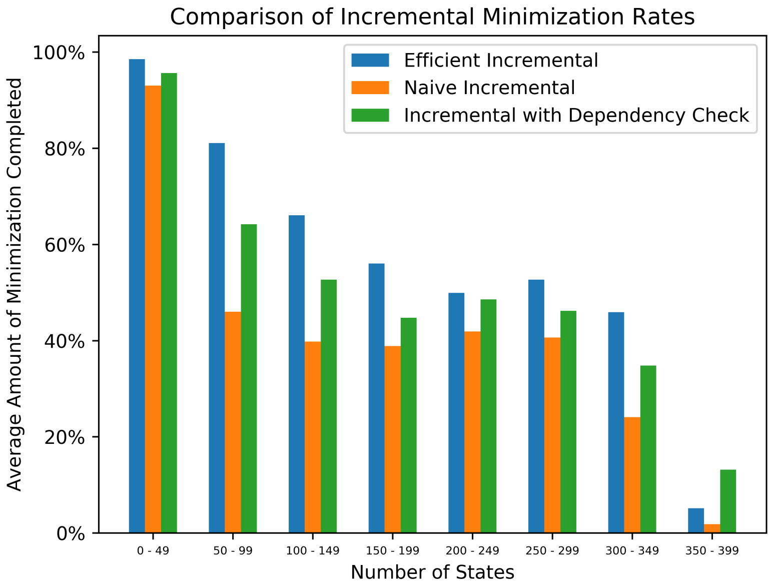

Figure 5.4: Average percentage of minimization completed in the same time span as the modified Hopcroft takes to finish minimization.

Our first metric for evaluating incremental performance will be the average percent of minimization completed in a given time span. Figure 5.4 graphs the results of running the incremental algorithms for exactly as long as the modified Hopcroft algorithm takes to complete minimization. Interestingly, the efficient incremental algorithm is able to complete at least 40% of minimization in the same time frame as the modified Hopcroft algorithm takes to complete execution in automata with under 350 states. However, there is a significant drop in performance for automata with between 350 and 400 states. Perhaps most surprising of all is the fact that the algorithm with dependency checking actually outperforms the efficient algorithm on average when given automata from this interval. It should be noted though that only five automata within our test set have more than 350 states. More experiments should be performed in the future to establish this trend.

Figure 5.4 represents the amount of minimization completed in a given amount of time. In contrast, Figure 5.5 displays the overall progress of incremental minimization. This figure is a heat map that graphs the average amount of time passed (as a fraction of the algorithm's total runtime) as a function of the percentage of minimization completed in that time and the number of states in the automaton. Blue/dark regions indicate that under 40% of the runtime of the algorithm has completed and yellow/light regions indicate that over 80% of the runtime of the algorithm has completed. For automata with a small number of states (under 100), the minimization appears to occur at a linear rate. For larger automata though, it is typical for a larger percentage of minimization to be completed in a smaller fraction of the algorithm's overall runtime. In particular, it appears to be common for over 60% of the automata to be minimized in only 40% of the runtime of the algorithm. The heat map only displays the performance of the efficient incremental algorithm. Similar figures could be constructed for the naive and dependency checking algorithms but it is generally difficult to compare the figures, so they were purposefully excluded.

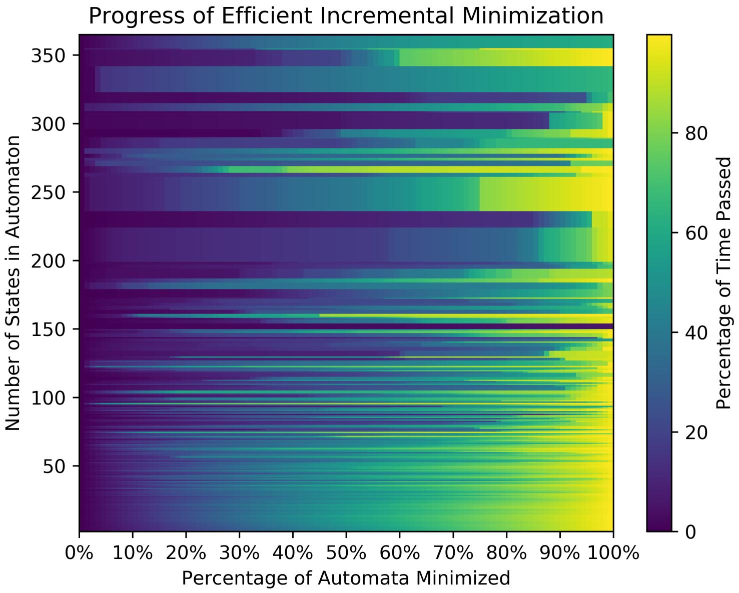

Figure 5.5: Heatmap displaying the average time required to reach a certain amount of minimization by the efficient incremental algorithm

5.4 Initialization of *neq*

Lastly, our initialization of *neq* in Algorithm 3 will be justified. The incremental algorithm given for DFAs in [2] initializes *neq* to contain only pairs of states where one state is accepting and the other is non-accepting. Instead, our algorithm initializes *neq* to contain all pairs of states that have varying minimum distances to an accepting state. This requires $O(m)$ precomputation, where m is the number of transitions in the automaton. A comparison of the average running times of these two initialization approaches, when both used as part of the incremental algorithm given in Section 3.2, is presented in Figure 5.6. From this figure, it can be seen that initializing *neq* to contain all pairs of states with different minimum distance to an accepting state generally has the best average runtime on symbolic automata.

Of the 838 automata tested with over 20 states, the algorithm with simple neq initialization only performed better in 39 cases. Even in these situations, the simple neq initialization was never observed to be more than 15 ms quicker (as opposed to a maximum of 473 ms slower) than the initialization based on distance to an accepting state.

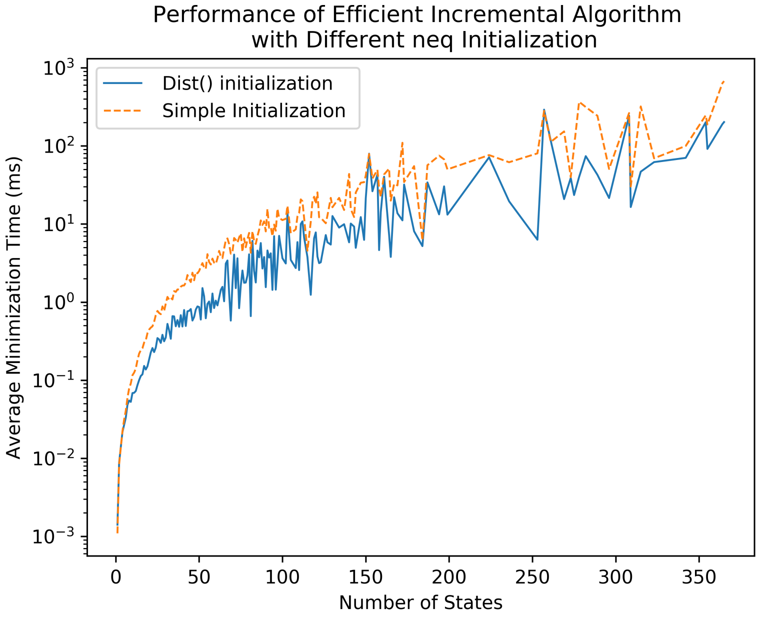

Figure 5.6: Performance of incremental algorithm with *neq* initialized only to pairs of accepting/non-accepting states compared to the same algorithm with *neq* initialized to all pairs with different minimum distance to an accepting state

Chapter 6

Conclusion

6.1 Related Work

Classical algorithms for the minimization of deterministic finite automata have been presented by Huffman [11], Moore [12], and Brzozowski [4]. Hopcroft's algorithm [10] for DFA minimization has the best worst-case time complexity of all known minimization algorithms. This algorithm runs in $O(kn \log n)$ time, where k is the alphabet size and n is the number of states in the automaton, and this bound is tight [3]. An algorithm for the incremental minimization of DFAs was first given by Watson [18]. However, the worst case performance of this algorithm is exponential. An efficient incremental algorithm was presented by Almeida, et al. in [1] (later republished as [2]). An incremental hybrid of the algorithms presented by Hopcroft and Almeida et al. is given in [9]. Extending this Hybrid algorithm to SFAs is an open problem.

The concept of automata with transitions defined by predicates was first conceived in [17] and first studied in [15]. Moore's algorithm for DFA minimization was initially adapted to symbolic automata in [16]. Hopcroft's algorithm was first adapted to symbolic automata in [6]. The minimization algorithms presented in [6] for deterministic SFAs were adapted to the computation of forward bisimulations for nondeterminstic symbolic automata in [7]. The minimization of symbolic transducers is also studied in [13]. An overview of the current state of research into the theory and application of symbolic automata and symbolic transducers is present in [8].

6.2 Conclusion and Future Work

An incremental algorithm for DFA minimization has been extended in several ways to the minimization of deterministic symbolic finite automata. Our most efficient algorithm runs in $O(n^6 f(2nl)\alpha(n))$ time. Although it generally performs worse than the efficient extension of Hopcroft's algorithm presented in [6], it has the advantage that the execution of the algorithm can be halted and an SFA with equivalent language to the input can be returned.

A similar incremental algorithm based on "dependency checking" was also introduced. However, it did not improve on the performance of the efficient algorithm experimentally. One future avenue of research is to continue the development of this dependency checking algorithm. Presently, it fails to improve on the runtime of the efficient algorithm because it only rarely is in a situation where all the dependencies of a given pair have been tested before some recursive call to `isEquiv` return false. Intuitively, one might extend this algorithm so that `isEquiv` does not halt when it reaches a pair of states that are provably distinguishable and instead, recursively considers all states until all dependencies have been tested. This approach would ensure that when `isEquiv` returns, all pairs tested in a recursive call would be provably distinguishable, or else necessarily equivalent. However, maintaining the incremental nature of the algorithm may be challenging.

Another possible avenue for the future development of these algorithms is the optimization of neq, the data structure used to store pairs of states known to be distinguishable. The algorithm presented in [2] initialized the neq set to contain only pairs of accepting and non-accepting states. This was optimized in Algorithm 3 for the symbolic setting by initializing the set to contain all pairs of states that have a different minimum distance to an accepting state. However, if we have already proven that states p and p' are equivalent and we have $(p', q) \in neq$, we do not necessarily have $(p, q) \in neq$ even though it must be true that p is distinguishable from q. There is clearly an opportunity for optimization here.

Lastly, an interesting possible extension of this algorithm would be to remove the assumption that the input automaton is deterministic and complete. Intuitively, this could be done by modifying the `isEquiv` function to test for equivalence on pairs of sets of states rather than pairs of states. Additionally, the process by which `isEquiv` generates local minterms would need to be altered to remove the assumption that every character in our domain is satisfied by some outgoing transition of every state.

Bibliography

[1] Marco Almeida, Nelma Moreira, and Rogério Reis. Incremental DFA minimisation. In *International Conference on Implementation and Application of Automata*, pages 39–48. Springer, 2010.

[2] Marco Almeida, Nelma Moreira, and Rogério Reis. Incremental DFA minimisation. *RAIRO-Theoretical Informatics and Applications*, 48(2):173–186, 2014.

[3] Jean Berstel and Olivier Carton. On the complexity of Hopcrofts state minimization algorithm. In *International Conference on Implementation and Application of Automata*, pages 35–44. Springer, 2004.

[4] Janusz A Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical theory of Automata*, 12(6):529–561, 1962.

[5] Loris D'Antoni. symbolicautomata. `https://github.com/lorisdanto/symbolicautomata/`. Accessed 2017-10-30.

[6] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *ACM SIGPLAN Notices*, volume 49, pages 541–553. ACM, 2014.

[7] Loris D'Antoni and Margus Veanes. Forward bisimulations for nondeterministic symbolic finite automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 518–534. Springer, 2017.

[8] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *International Conference on Computer Aided Verification*, pages 47–67. Springer, 2017.

[9] Pedro García, Manuel Vázquez de Parga, Jairo A Velasco, and Damián López. A split-based incremental deterministic automata minimization algorithm. *Theory of Computing Systems*, 57(2):319–336, 2015.

[10] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

[11] David A Huffman. The synthesis of sequential switching circuits. *Journal of the franklin Institute*, 257(3):161–190, 1954.

[12] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[13] Olli Saarikivi and Margus Veanes. Minimization of symbolic transducers. In *International Conference on Computer Aided Verification*, pages 176–196. Springer, 2017.

[14] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[15] Gertjan van Noord and Dale Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286, 2001.

[16] Margus Veanes, Peli De Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 498–507. IEEE, 2010.

[17] Bruce W Watson. Implementing and using finite automata toolkits. *Natural Language Engineering*, 2(4):295–302, 1996.

[18] Bruce W Watson. An incremental dfa minimization algorithm. In *International Workshop on Finite-State Methods in Natural Language Processing, Helsinki, Finland*, 2001.

Appendix A

isEquiv Souce Code

```
class EquivTest{

   protected class EquivRecord{
      public final Integer pState;
      public final Integer qState;
      public final HashSet <List<Integer>> curPath;

      public EquivRecord(Integer p, Integer q, HashSet<List<Integer>> curPath){
         this.pState = p;
         this.qState = q;
         this.curPath = curPath;
      }
    }

   protected final DisjointSets<Integer> equivClasses;

   protected HashSet<List<Integer>> equiv;
   protected HashSet<List<Integer>> path;

   public EquivTest(DisjointSets<Integer> equivClasses,
         HashSet<List<Integer>> equiv,
         HashSet<List<Integer>> path){
      this.equivClasses = equivClasses;
      this.equiv = equiv;
      this.path = path;
   }

   protected List<SFAInputMove<P,S>> findNonDisjointMoves(
         Collection<SFAInputMove<P, S>> outp,
         Collection<SFAInputMove<P, S>> outq) {
        /* Analogous to the operation on line 10 of algorithm 5.
        Similar to the implementation in [6], we need not actually
        find a witness a to the constructed local minterm */
      SFAInputMove<P,S> pMove = outp.iterator().next();
      P pGuard = pMove.guard;
      for(SFAInputMove<P,S> qMove : outq){
         P qGuard = qMove.guard;
         P pqAnd = ba.MkAnd(pGuard, qGuard);
        if(ba.IsSatisfiable(pqAnd)){
            return Arrays.asList(pMove,qMove);
        }
    }
    return null; //never reached assuming SFA is deterministic/complete
}

public boolean isEquiv(Integer pStart, Integer qStart){
    if (isKnownNotEqual(pStart,qStart)){
        return false;
    }
    EquivRecord start = new EquivRecord(pStart,qStart,path);
    Stack<EquivRecord> testStack = new Stack<EquivRecord>();
    testStack.add(start);
    while (!testStack.isEmpty()){
        EquivRecord curEquivTest = testStack.pop();
        Integer p = curEquivTest.pState;
        Integer q = curEquivTest.qState;
        HashSet<List<Integer>> curPath = curEquivTest.curPath;
        List<Integer> pair = normalize(p,q);
        HashSet<List<Integer>> newPath = new HashSet<List<Integer>>(curPath);
        newPath.add(pair);
        Collection<SFAInputMove<P,S>> outp =
                new ArrayList<SFAInputMove<P,S>>(aut.getInputMovesFrom(p));
        Collection<SFAInputMove<P,S>> outq =
                new ArrayList<SFAInputMove<P,S>>(aut.getInputMovesFrom(q));
        while(!outp.isEmpty() && !outq.isEmpty()){
            List<SFAInputMove<P,S>> nonDisjointGuards =
                    findNonDisjointMoves(outp, outq);
            SFAInputMove<P,S> pMove = nonDisjointGuards.get(0);
            SFAInputMove<P,S> qMove = nonDisjointGuards.get(1);
            Integer pNextClass = equivClasses.find(pMove.to);
            Integer qNextClass = equivClasses.find(qMove.to);
            List<Integer> nextPair = normalize(pNextClass, qNextClass);
            if(!pNextClass.equals(qNextClass) && !equiv.contains(nextPair)){
                if(isKnownNotEqual(pNextClass,qNextClass)){
                    this.path = newPath;
                    return false;
                }
                if (!newPath.contains(nextPair)){
                    equiv.add(nextPair);
                    EquivRecord nextTest =
                            new EquivRecord(pNextClass, qNextClass, newPath);
                    testStack.push(nextTest);
                }
            }
            outp.remove(pMove);
            outq.remove(qMove);
            P newPGuard = ba.MkAnd(pMove.guard, ba.MkNot(qMove.guard));
            if (ba.IsSatisfiable(newPGuard)){
                outp.add(new SFAInputMove<P,S>(pMove.from, pMove.to, newPGuard));
            }
            P newQGuard = ba.MkAnd(qMove.guard, ba.MkNot(pMove.guard));
            if (ba.IsSatisfiable(newQGuard)){
                    outq.add(new SFAInputMove<P,S>(qMove.from, qMove.to, newQGuard));
                }
            }
        }
        equiv.add(normalize(pStart, qStart));
        return true;
    }
}
```