International Journal of Foundations of Computer Science
Vol. 20, No. 4 (2009) 735–746
© World Scientific Publishing Company

HYPER-MINIMIZATION IN $O(n^2)$

ANDREW BADR
6506 Wheeler St.
Oakland, CA 94609
andrewbadr@gmail.com

Received 24 May 2009
Accepted 25 June 2009
Communicated by Oscar H. Ibarra and Balasubramanian Ravikumar

Two formal languages are *f-equivalent* if their symmetric difference $L_1 \triangle L_2$ is a finite set — that is, if they differ on only finitely many words. The study of f-equivalent languages, and particularly the DFAs that accept them, was recently introduced [3]. First, we restate the fundamental results in this new area of research. Second, we introduce our main result, which is a faster algorithm for the natural minimization problem: given a starting DFA D, find the smallest (by number of states) DFA D' such that $L(D)$ and $L(D')$ are f-equivalent. Finally, we suggest a technique that combines this *hyper-minimization* with the well-studied notion of a *deterministic finite cover automaton* [2, 4, 5], or DFCA, thereby extending the applicability of DFCAs from finite to infinite regular languages.

Keywords: DFA; DFCA; minimization; hyper-minimization.

1991 Mathematics Subject Classification: 68Q05, 68Q68

1. Introduction, Notation, and Prior Results

The now-classical notions of DFA *equivalence* and *minimization* are well-studied [6, 1]. Two DFAs D_1 and D_2 are *equivalent* if the languages they induce are equal. We write this as $D_1 \equiv D_2$. In the recently-introduced study of *f-equivalence* [3][a], this condition is loosened: instead of requiring that the languages be equal, one allows them to differ by finitely many words.

In this paper, we continue the study of *f-equivalence.* One motivation is that the *preamble* and *kernel* partition increases our general understanding of DFAs, and may find use in other algorithms. *Hyper-minimization*, and the newly introduced *finite-factoring*, may be useful in some settings in reducing the size of finite automata. As for notation, we use the standard definition of a DFA as a 5-tuple $(Q, \Sigma, \delta, q_0, A)$ where Q is the state-set, Σ is the alphabet, δ is the extended transition function, q_0 is the starting state, and A is the accepting subset of Q. For more

[a]There, f-equivalence is called either "almost equivalence" or "finite difference". We use the new term here because it is shorter, and cannot be misunderstood as excluding total equivalence.

on DFAs, see any standard reference [6, 1]. In all algorithm analyses, "n" implicitly refers to the number of states of the DFA in question. Where it is unspecified, L is assumed to be a language, D a DFA, and q a state. These may be subscripted when two or more languages, automata, or states are being considered simultaneously. Finally, components that share a subscript such as Q_1, δ_1 etc. should be assumed to be part of a DFA D_1.

Definition 1 (f-equivalence) *Two languages L_1 and L_2 are said to be* f-equivalent *if $L_1 \triangle L_2$, their symmetric difference, is a finite set. We write $L_1 \sim L_2$. This relation is extended to DFAs in the obvious way: if $L(D)$ is the language recognized by a DFA D, then we write $D_1 \sim D_2$ whenever $L(D_1) \sim L(D_2)$. Finally, f-equivalence can also be considered on DFA states. States q_1 and q_2 are f-equivalent ($q_1 \sim q_2$) if their induced languages (sometimes called right-languages) are f-equivalent ($L(q_1) \sim L(q_2)$). States q_1 and q_2 need not be in the same DFA.*

Many interesting features of the f-equivalence relation have been discovered. In this section, we restate and explain the most important of these to the reader (since these ideas are still new). One should, however, refer to the original paper for analyses.

Like classical equivalence, *f-equivalence* can be seen as an equivalence relation on either the languages themselves (we write $L_1 \sim L_2$) or the DFAs recognizing them ($D_1 \sim D_2$). DFA f-equivalence (like classical-equivalence) is an equivalence-relation, so it partitions the set of all DFAs into equivalence-classes. Since two classically-equivalent DFAs are also trivially f-equivalent, the classical-equivalence partition is a refinement of the f-equivalence partition.

We begin with two trivial but very useful results. First:

Proposition 2. *Let q_1 be a state from DFA D_1, and q_2 be a state from D_2. If $q_1 \sim q_2$, then for any input c: $\delta_1(q_1, c) \sim \delta_2(q_2, c)$.*

Notice that this is directly analagous to a statement about classical equivalence. Second:

Corollary 3. *If $D_1 \sim D_2$, then $(\forall q_1 \in Q_1, \exists q_2 \in Q_2, : q_1 \sim q_2)$.*

Again, this is directly analagous to a statement for classical equivalence: if two DFAs are equivalent, their states are members of precisely the same set of Myhill-Nerode equivalence classes.

Next, we define a partition of every DFA's state set. This partition turns out to be critical to the study of f-equivalence:

Definition 4 (Preamble and Kernel) *For any DFA $D = (Q, \Sigma, \delta, q_0, A)$, Q is partitioned into the* preamble *and* kernel *parts: $P(D)$ and $K(D)$. A state q is in the preamble $P(D)$ if its left-language is finite — that is, if there are only finitely many strings w such that $\delta(q_0, w) = q$ — and in the kernel otherwise. In short, the states are divided according to whether they are reachable from q_0 by only finitely many or by infinitely many strings.*

Finally, we restate the f-equivalence isomorphism and minimality results. Once again, we emphasize that the interested reader should refer to the original for proofs [3]. The results are presented here primarily as background, and also to give these ideas wider exposure.

Definition 5 (Kernel Isomorphism) *Given DFAs $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, A_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, A_2)$, we say that D_1 and D_2 have isomorphic kernels (and write $D_1 \cong_K D_2$) when there exists a bijection $f : K(D_1) \to K(D_2)$ such that*

1. $\forall q_1 \in K(D_1) : q \in A_1 \Leftrightarrow f(q) \in A_2$ and
2. $\forall q_1 \in K(D_1), \forall c \in \Sigma : f(\delta_1(q_1, c)) = \delta_2(f(q_1), c)$.

Theorem 6 (Kernel Isomorphism) *If $D_1 \sim D_2$ and both automata are classically minimized, then $D_1 \cong_K D_2$*

Note that the converse of Theorem 6 is false. It is possible that two automata with isomorphic kernels are not f-equivalent. For example, consider the automata recognizing the languages 0^* and 10^*.

Definition 7 (Hyper-minimality) *A DFA $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, A_1)$ is called* hyper-minimized *if for any DFA $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, A_2)$, it holds that $(D_1 \sim D_2) \Rightarrow (|Q_1| <= |Q_2|)$.*

Theorem 8 (Characterizing Hyper-minimality) *A DFA $D = (Q, \Sigma, \delta, q_0, A)$ is hyper-minimal if and only if:*

1. D is classically minimized, and
2. $\forall q_1 \in Q, \forall q_2 \in (Q - \{q_1\}) : (q_1 \sim q_2) \Rightarrow (q_1 \in K(D) \vee q_2 \in K(D))$.

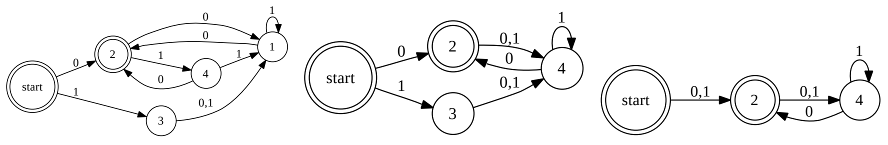
Definition 9 (Preamble Isomorphism) *Given DFAs $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, A_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, A_2)$, we say that D_1 and D_2 have isomorphic preambles (and write $D_1 \cong_P D_2$) when there exists a bijection $f : P(D_1) \to P(D_2)$ such that: $\forall q_a \in P(D_1), \forall q_b \in P(D_1), \forall c \in \Sigma : \delta_1(q_a, c) = q_b \to \delta_2(f(q_a), c) = f(q_b)$.*

The definition of preamble isomorphism is weaker than kernel isomorphism because f does not preserve acceptance (membership in A).

Theorem 10 (Preamble Isomorphism) *If $D_1 \sim D_2$ and both automata are hyper-minimized, then $D_1 \cong_P D_2$*

Notice that Theorem 10 requires that the automata are hyper-minimized while Theorem 6 only requires them to be classically minimized.

To conclude this section, we briefly note that these two isomorphism theorems are optimal in the sense that any aspect of the DFA that they do not preserve can indeed vary between f-equivalent and hyper-minimized automata. These unfixed parameters are the start state q_0, which can be moved with an f-equivalence class; acceptance in the preamble ($P(D) \cap A$), which can be changed arbitrarily; and transitions leading from the preamble to the kernel, the destinations of which can move within an f-equivalence class.



(a) Original DFA (b) Minimized (c) Hyper-Minimized

Fig. 1. Example of hyper-minimization. The DFA's language changes on the word "1". The state labeled "3" is merged into "2", leaving on "start" in the preamble.

2. Hyper-Minimization Algorithm

2.1. *Algorithm overview*

The problem of hyper-minimization is a fundamental part of the study of f-equivalence, and perhaps the most strongly motivated. Given a starting DFA D, we seek the smallest D' (by number of states) such that $D \sim D'$. (Note that this can, but doesn't always eliminate the preamble). Here, we present a new hyper-minimization algorithm, which is the fastest yet known. The original algorithm ran in $O(n^3)$-time; this one runs in $O(n^2)$. Furthermore, this algorithm is more direct, involving no iterative partition refinement, and uses the perhaps surprising technique of constructing the cross-product of a DFA with itself.

We begin with a top-down sketch of hyper-minimization, then explain the components from the bottom up. Both the new and the original hyper-minimization algorithms share the following highest-level structure:

Algorithm 11 (hyper-minimize)
Input: a starting DFA $D = (Q, \Sigma, \delta, q_0, A)$
Output: a hyper-minimized version of D

```
1. Let $D' = minimize(D)$, where 'minimize' is classical DFA minimization
```

2. Let E = $f\_equivalence\_classes(D')$ be the partition of Q' into f-equivalence classes. This may be obtained via Algorithm 14 (Main Result) below.
3. Let $P, K = preamble\_and\_kernel(D')$ be the preamble and kernel subsets of Q'. This may be obtained via Algorithm 15 below.
4. $f\_merge\_states(D', E, P, K)$, for example via Algorithm 16 below. This is the operation of merging states within each f-equivalence class.
5. Return D'

Because the new and original algorithms share this outline, we refer to the original for proof that it is valid [3]. As with classical minimization, the meat of the problem is in finding the state equivalence classes (Step 2), which is the only step that here differs from the original paper. Merging states (Step 4) is similar to the step from classical DFA minimization, but extra consideration must be made of the preamble and kernel.

We claim that the four steps of Algorithm 11 can collectively be executed in quadratic time. Step 1, famously, can be accomplished in $O(n * log(n))$ time [1] and requires no further explanation. Step 2 is explained in more detail as Algorithm 14. We prove below that it can be accomplished in time $O(n^2)$. For full explanations and analyses of Steps 3 and 4, the reader is referred to the original paper [3], though we do offer an overview below.

Implementations of all the algorithms in this paper are available in the Python programming language at http://ianab.com/hyper/

2.2. *Algorithm details*

The above hyper-minimization outline is roughly analagous to one for classical DFA minimization:

1. Remove all unreachable states
2. Partition the states into Myhill-Nerode equivalence-classes
3. Collapse each equivalence class into a representative state

All DFA minimization algorithms save one fit into this framework [8]. As in hyper-minimization, the meat of the problem is in partitioning the states into equivalence classes, with the other steps being quite straightforward in comparison. (One difference is that the collapsing of classes is more complicated under hyper-minimization, requiring the computation of the kernel and preamble.)

We will now work up towards Step 2 of Algorithm 11, presenting and analyzing the new method by which a partition into f-equivalence classes can be accomplished in time $O(n^2)$. This is our main result. Afterwards, we will discuss Steps 3 and 4 of Algorithm 11.

Our algorithm will use the following version of the standard cross-product DFA construction [1]. The standard construction is usually presented for constructing an automaton recognizing the union or intersection of two others. Here, we use the exclusive-or operator instead.

Definition 12 (xor_cross_product) *Given DFAs* $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, A_1)$ *and* $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, A_2)$*, define* $xor\_cross\_product(D_1, D2) = D^{\otimes} = (Q^{\otimes}, \Sigma, \delta^{\otimes}, q_0^{\otimes}, A^{\otimes})$ *as follows:*

1. *Let* $Q^{\otimes} = \{(q_1, q_2) : q_1 \in Q_1 \wedge q_2 \in Q_2\}$
2. $\forall q_1 \in Q_1, \forall q_2 \in Q_2, \forall c \in \Sigma :$ *Let* $\delta^{\otimes}((q_1, q_2), c) = (\delta_1(q_1, c), \delta_2(q_2, c))$
3. *Let* $q_0^{\otimes} = (q_{0,1}, q_{0,2})$
4. *Let* $A^{\otimes} = \{(q_1, q_2) : (q_1 \in A_1) \otimes (q_2 \in A_2)\}$ *where* $\otimes$ *is the xor operation*

Note that the three DFAs share the same alphabet Σ. See Figure 2 for an example.

Algorithm 13 (right_finite_states)
Input: a DFA $D = (Q, \Sigma, \delta, q_0, A)$*, and the set* S *of all states in* Q *that induce the* empty *language (that is,* $S = \{q \in Q : \forall w \in \Sigma^* : \delta(q, w) \notin A\}$*).*
Output: the subset $F \subset Q$ *of all states that induce a* finite *language*
Running-time: $O(n)$

```
1. Let $S'$ be the complement of $S$
2. For each state $q$ let $Incoming_q$ and $Outgoing_q$ be new empty sets
3. For each $q \in S'$: for each $c \in \Sigma$:
    (1) Let $q' = \delta(q, c)$
    (2) Add $(q, c)$ to the set $Incoming_{q'}$
    (3) Add $(q', c)$ to the set $Outgoing_q$
4. Let $F$ be a new empty list
5. Let to_process be a new list equal to $S$
6. While to_process is nonempty:
    (1) Let $q = pop(to\_process)$
    (2) Add $q$ to $F$
    (3) For each $(q', c) \in Incoming_q$:
        i. Remove $(q, c)$ from $Outgoing_{q'}$
        ii. If $Outgoing_{q'}$ is now empty, add $q'$ to to_process.
7. Return $F$
```

Algorithm 13. We seek to prove first that the algorithm is correct, and second that it runs in linear time with respect to Q.

Correctness: When a state is added to F in the processing loop, we call it "removed" from the DFA. This algorithm removes every state in the "sink-set" S, then (while any such state exists) removes all states such that all the state's outgoing transitions lead to removed states. It remains to prove that a state is removed if and only if it induces a finite language.

First, we prove that if a state is removed, then it induces a finite language. Note that there are no transitions from S to a state in S'. Otherwise, the state in S' would also induce an empty language, so by assumption it would be in S. Next, assign to each removed state q a distance $d(q)$ from S, equal to the length of the longest path from q to a state in S. We obtain the result by induction on d. If $d(q) = 0$, then $q \in S$ and $|L(q)| = 0$ by definition. The size of $L(q)$ for any state q in a DFA is bounded above by $1 + \Sigma_{q' \in Outgoing_q} L(q')$. Therefore, if all removed states with $d <= n$ induce finite languages, it follows that all states with $d = n + 1$ also induce a finite language, completing this direction.

Second, we prove that if a state q induces a finite language, it is removed by the algorithm. Again we use a simple inductive proof. Let $l(q)$ be the length of some longest word w in $L(q)$. If $l(q) = 0$ then q is in S, so it is removed by the algorithm. If $l = n$, then for every state $q' \in Outgoing_q$, $l(q') < l(q)$. Therefore, if every state with $l <= n$ is removed by the algorithm, then every state with $l = n + 1$ is also removed, because all states it transitions to are removed.

Time Complexity: Building the *Incoming* and *Outgoing* sets takes linear time because it takes a constant amount of time for each transition and the number of transitions is linear with the number of states. Removing any state takes constant time (for popping it from *to_process* and adding it to F, plus some amount of work for each incoming transition). Again, there are only $O(n)$ transitions, so the latter part adds up to work linear in the number of states. These steps compose the algorithm. □

In the following algorithm, the input DFA is assumed to be minimized. This is purely so that we may omit minimization as an unilluminating first step. The algorithm also refers to a Union-Find data structure. This data structure allows one to combine elements into disjoint sets, and find which set an element belongs to[7].

Algorithm 14 (f_equivalence_classes (Main Result))
Input: a minimized DFA $D = (Q, \Sigma, \delta, q_0, A)$
Output: a partition of Q (the state-set of D) into the equivalence-classes determined by the f-equivalence relation (the "f-equivalence classes")
Running-time: $O(n^2)$

1. *Let $D^{\otimes} = xor\_cross\_product(D, D)$*
2. *Let $S = \{(q, q) : q \in Q\}$ be the set of all self-pair states in $D^{\otimes}$. These are highlighted in Figure 2.*
3. *Let $F = right\_finite\_states(D^{\otimes}, S)$ be the set of all states (q, r) such that (q, r)*

```
   induces a finite language in $D^{\otimes}$. (Algorithm 13)
4. Use the state-pairs in $F$ to construct a partition $P$ of $Q$:
   (1) Let $P$ be a new Union-Find data structure
   (2) For each state $q \in Q$: make a new set $\{q\}$ in $U$
   (3) For each $(q, r)$ in $F$:
         i. Let $P_q = P.find(q)$
        ii. Let $P_r = P.find(r)$
       iii. If $P_r \neq P_q$, then $P.union(P_q, P_r)$
5. Return $P$
```

Algorithm 14.
Correctness: For every word $w \in \Sigma^*$ and state $(q, r) \in Q^{\otimes}$ we have $w \in L((q, r)) \Leftrightarrow \delta^{\otimes}((q, r), w) \in A^{\otimes} \Leftrightarrow (\delta(q, w) \in A) \otimes (\delta(r, w) \in A)$ by definition. In other words, the language $L((q, r))$ of every state in the $D^{\otimes}$ context equals the language $L(q) \triangle L(r)$ in the D context. The first consequence of this is that S is exactly the set of states in $D^{\otimes}$ that induce the empty language, because the given DFA D is minimized (so no two distinct states induce the same language). This proves that the input to *right_finite_states* is correct, and the result F is as desired. Thus, as a second consequence of the above, we see that $L((q, r))$ is finite if and only if $q \sim r$. Therefore, F is the f-equivalence relation on the states in D. Step 4 turns this relation, represented as a set of pairs, into a partition.

Time Complexity: The DFA cross-product construction in Step 1 clearly takes $O(n^2)$ time. Constructing S in Step 2 clearly takes $O(n)$ time. In Step 3, we construct F using *right_finite_states* (Algorithm 13), which was proven to take time linear in the number of states. Since the input DFA has n^2 states, Step 3 takes time $O(n^2)$. In Step 4, we iterate through $O(n^2)$ pairs and do an equivalent number of Find operations. Since there are only n states, at most $n - 1$ Union operations are performed. Therefore, by using a Union-Find data-structure that has constant-time Find and linear-time Union [7], this step also takes $O(n^2)$ time. □

This concludes the main result of the paper. We now continue with Steps 3 and 4 from Algorithm 11. Once again, since these are exactly the same as in the original paper[3], the reader is directed there for additional analysis.

Algorithm 15 (preamble_and_kernel)
Input: a DFA $D = (Q, \Sigma, \delta, q_0, A)$
Output: a pair of sets, the first containing the preamble states of D, and the second containing the kernel states of D
Running-time: $O(n^2)$

```
1. Let $K$ be an empty set
2. For each $q \in Q$: let $R_q$ be the set of states nontrivially reachable from $q$
```

```
3. For each $q \in Q$: if $q \in R_q$: Let $K = K \cup R_q$
4. Return $(Q - K, K)$
```

Algorithm 16 (f_merge_states)
Input: a minimized DFA $D = (Q, \Sigma, \delta, q_0, A)$, the partition E of its states into f-equivalence classes, and the partition (P, K) of its states into the preamble and kernel
Output: D is hyper-minimized
Running-time: $O(n)$

```
1. For each set $S$ in $E$:
    (1) Let $P_S = S \cap P$
    (2) Let $K_S = S \cap K$
    (3) If $K_S$ is non-empty: Let $R = pop(K_S)$
    (4) Else: Let $R = pop(P_S)$
    (5) For each state $q$ in $P_S$: merge $q$ into $R$
```

The "merge" in the final step above refers to a procedure that is familiar from classical minimization: all transitions to the first state are redirected to the second state, and the first state is deleted from the automaton.

3. Finite-Factoring with the DFCA

In some sense, hyper-minimization pares down a regular language to its core. Outlier words are added or removed to make the DFA as small as possible. However, in some circumstances, one may want to keep track of exactly *which* words were changed in the course of hyper-minimization. Such a list is not difficult to obtain: if D is the original DFA and D' is the hyper-minimized version, the xor_cross_product of D and D' recognizes precisely the finite difference that was changed. (A more complicated algorithm could keep track of which words change during the minimization process.)

The DFCA, or deterministic finite cover automaton, is a fairly well-studied [2, 4, 5] variation on the classical DFA. A DFCA can save space in recognizing finite languages, in proportion to their redundancy, essentially by removing the need of the DFA to keep track of the length of the string. A DFCA C can be understood as a pair $C = (D, l)$ where D is a DFA and l is a non-negative integer. Now C accepts a word w if both $|w| < l$ and D accepts w (where $|w|$ is the length of w). Minimization algorithms have been given [4] that, given a starting DFA D_1 and maximum desired length l, can quickly reduce the DFA to minimal DFCA $C = (D_2, l)$ that agrees with D_1 on all words of length less than l.

An obvious weakness of the DFCA is that its ability to remove the redundant computation necessitated by counting can only be applied to finite languages. By

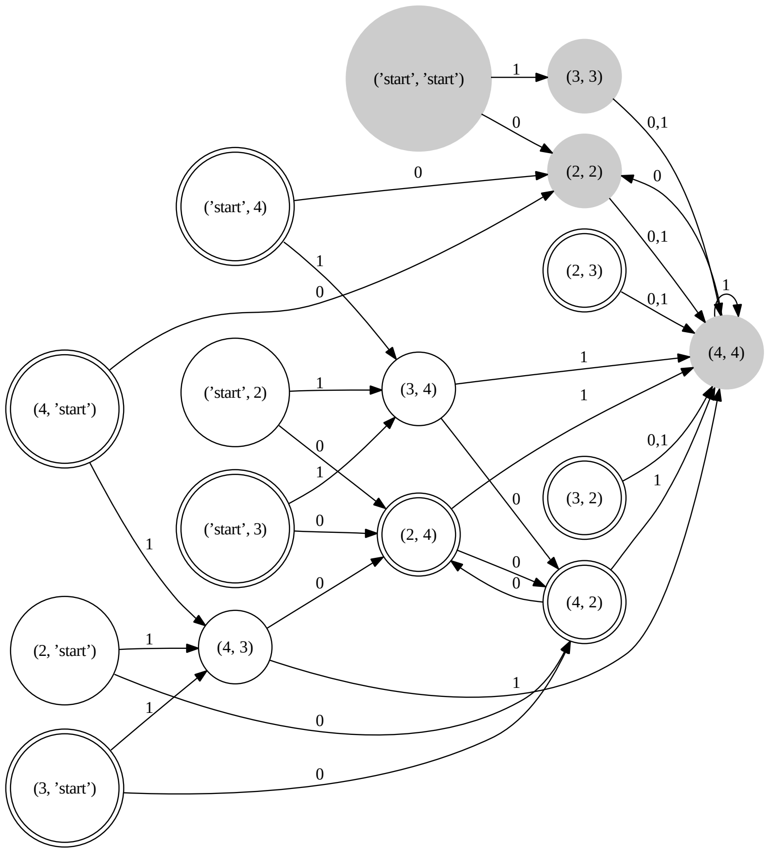


Fig. 2. The minimized DFA of Figure 1 crossed with itself, as in Step 1 of Algorithm 14. The self-cross states, shaded dark, are exactly the sink states of this automaton. The set of right-finite states will reveal the f-equivalence relation.

combining hyper-minimization with the DFCA, this weakness can for the first time be overcome.

Definition 17 (Finite-Factored Automaton) *A* finite-factored automaton *is a pair (D, C) where the first item is a DFA and the second is a DFCA. A finite-factored automaton accepts a word w if and only if exactly one of D and C accepts w.*

Algorithm 18 (finite-factor)
Input: a DFA D
Output: a finite-factored automaton pair (D', C), where D' is a DFA and C a DFCA, such that for all words w, D accepts w if and only if (D', C) accepts w.
Running-time: $O(n^2 * log(n))$

```
1. Let $D' = hyper\_minimize(D)$
```

```
2. Let $D_f = xor\_cross\_product(D, D')$
3. Let $l = max(|w| : w \in L(D_f))$
4. Minimize the DFCA $(D_f, l)$ [4]
5. Return $(D', (D_f, l))$
```

It is clear from a simple example that finite-factoring can greatly reduce the number of states required to recognize a regular language. Consider a language over $\Sigma = \{0, 1, a, b, c, d, e\}$ that accepts a word w if w contains only numbers and is up to nine characters long, or if w contains only letters (of any length). This language L requires twelve states to represent with a minimized DFA, and a DFCA cannot be used directly because L is infinite. However, finite-factoring results in two states for the DFA, and three states for the DFCA, for a total reduction of seven states. (See Figure 3 for factored components.)

This reduction seems to have been possible because L contains a finite-sized subset of words that, when considered as their own language, would induce a DFA are amenable to reduction with a DFCA. Hyper-minimization was be used to isolate this component from the rest of L, so that the DFCA reduction could be applied. Analysis of this technique is left for future research.

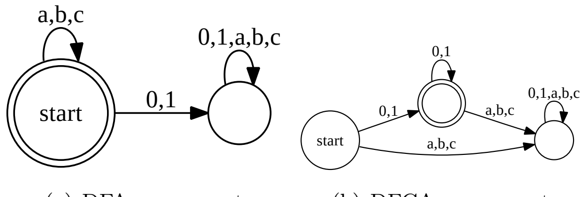


Fig. 3. Example of finite-factoring from text, above. The results from these two components will be xor'd.

4. Conclusion and Open Problems

The question we address in this paper — in short, "What can be said about f-equivalent automata?" — is a quite natural one. However, it has (until recently) gone unaddressed in the now half-century-long study of DFAs. In this paper, we reviewed the fundamental results in this new area, then provided a significantly improved algorithm for the central problem of hyper-minimization. We conclude with a few open problems:

1. DFA minimization is famously solvable in time $O(n * logn)$. DFCA minimization, too, was quickly reduced from $O(n^4)$ in the original paper [2] to an $O(n * logn)$ algorithm [4]. Can hyper-minimization also be achieved in $O(n * logn)$?

2. There are numerous open problems surrounding finite-factored automata. For example: does the method presented here always result in the smallest total number of states? If not, are some hyper-minimized automata in the same equivalence class better than others? Is there an asymptotically faster algorithm?
3. Starting with a minimized DFA, first change the acceptance values of selected preamble states, then minimize the DFA again. It is clear that for some automata, this process can produce a hyper-minimized result. For exactly which automata is this true?

References

[1] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computability*, (Addison-Wesley Longman Publishing Co., Inc, Boston, 2000).

[2] C. Câmpeanu, N. Santean and S. Yu, Minimal Cover-Automata for Finite Languages, in *"Workshop on Implementing Automata 1998"*, pp.43–56.

[3] A. Badr, V. Geffert and I. C. Shipman, Hyper-minimizing minimized deterministic finite state automata, *RAIRO - Theoretical Informatics and Applications*. **43**(2009) 69–94, doi:10.1051/ita:2007061.

[4] H. Körner, A time and space efficient algorithm for minimizing cover automata for finite languages, *International Journal of Foundations of Computer Science*, **14**(2)(2003) 1071–1086.

[5] C. Câmpeanu, A. Paun, J. R. Smith, Incremental construction of minimal deterministic finite cover automata, *Theor. Comput. Sci.*, **363**(2)(2006) 135–148.

[6] M. Sipser, *Introduction to the Theory of Computation*, 2nd edn. (International Thomson Publishing, 1996).

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest and Cliff Stein, *Introduction to algorithms*, (MIT Press, Cambridge, MA, USA, 2001).

[8] B. W. Watson, A taxonomy of finite automata minimization algorithms, *Computing Science Report 93/44*, (Eindhoven University of Technology, Eindhoven, The Netherlands, 1993).