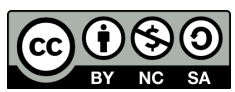
CDM

Fast Minimization

Klaus Sutner

Carnegie Mellon University

Fall 2024



Minimization is a good example where efficient computation forces one to think more carefully than in math alone.

We have a quadratic time (worst case), linear space algorithm that works fairly well as long as the machines are not too large. If one tries to break through the quadratic barrier, a reasonable target is log-linear.

So the challenge is to come up with a log-linear algorithm for minimization, or come up with plausible arguments as to why such an algorithm does not exist. To be clear: it does exist, but it's much more complicated than Moore's algorithm.

Mathematical Thinking: behavioral equivalence. Once the concept of behavior is clear, there is a straightforward algorithm for minimization. And, it's even polynomial time.

Algorithmic Thinking: refinement of equivalence relations. A better algorithm is obtained by thinking clearly about computing with equivalence relations (Moore). The reward is a clean, quadratic time algorithm (which is often much better than quadratic).

Smart Algo Thinking: baby-steps vs. giant-steps. Now things get tricky: all sub-quadratic algorithms require a much more careful argument and deeper algorithmic methods. A bit of creative insight is required to get down to log-linear. And doing things elegantly and efficiently is quite difficult.

Recall our abstract scenario: we have an equivalence relation $\rho \subseteq Q \times Q$ and an endofunction $f : Q \to Q$.

We want to find the coarsest refinement $\widehat{\rho}$ of ρ that is compatible with f:

$$p \mathrel{\widehat{\rho}} q \Rightarrow f(p) \mathrel{\widehat{\rho}} f(q)$$

This is accomplished by repeated application of a refinement operator ref_f:

$$p \mathrel{\rho_f} q \Leftrightarrow f(p) \mathrel{\rho} f(q)$$

$$\mathsf{ref}_f(\rho) = \rho \sqcap \rho_f$$

In other words: $\widehat{\rho}$ is the fixed point of ρ under ref_f.

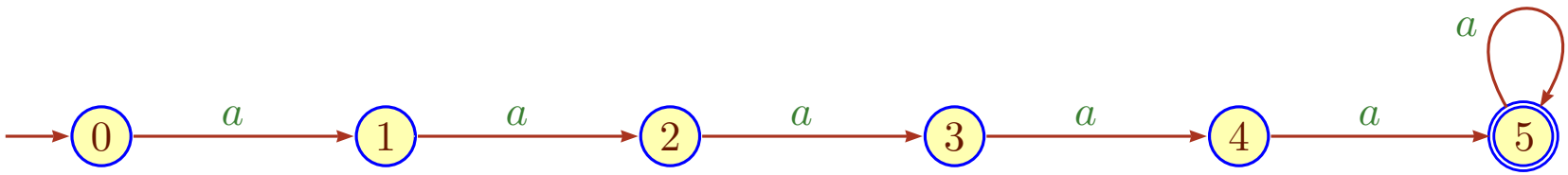
The refinement operator in Moore's algorithm works by representing all relations as canonical selector functions (aka int arrays), each round requires a scan of the whole array.

So each refinement step is $\Theta(n)$—with good constants but still linear in n.

The good news is that, quite often, the algorithm uses fewer than n rounds, so the total time complexity may well be sub-quadratic.

Alas, there are cases when Moore requires $\Theta(n)$ rounds, producing quadratic running time overall.

Here is the standard example that demonstrates that Moore's algorithm may be quadratic: the minimal DFA for $\{a\}^{\geq k}$.



For this automaton, a single Moore round will split off only one state from the right end of block $D = \{0, 1, \ldots, r\}$, at a cost of $\Theta(n)$ steps.

The split occurs only because of some block $B = \{p\}$, nothing else matters.

Critical Idea:
Maybe we could get mileage out of trying to guide the refinement by single blocks, instead of blindly hitting the whole carrier set.

We only want to touch the the blocks in the partition that are currently "relevant," not just blindly every state in the machine. E.g., in the last machine we want to touch only $\{0, \ldots, p-1\}$.

Suppose ρ is a partition of Q, and consider two blocks C and B. Let $f : Q \to Q$ be some endofunction.

We say that C splits B if

$$B \cap f^{-1}(C) \neq \emptyset \quad \text{and} \quad B - f^{-1}(C) \neq \emptyset.$$

In other words, $f(B)$ intersects both C and $Q - C$ and is not f-compatible yet: stopping the refinement process at this point would produce a nondeterministic machine. We need further refinement.

Let's define a new, more complicated refinement operator $\rho' = \mathrm{ref}_f(\rho, B, C)$ as follows:

$$p \; \rho' \; q \iff (p, q \notin B \wedge p \; \rho \; q) \vee$$

$$\Big(p, q \in B \wedge (f(p) \in C \Leftrightarrow f(q) \in C)\Big)$$

In other words: outside of block B we keep the old ρ. Inside of B we check for C-equivalence of children.

So $\mathrm{ref}_f(\rho, B, C)$ is a indeed a refinement of ρ: block B is potentially split in two (or may be unchanged).

Proposition

- $\mathsf{ref}_f(\rho) \sqsubseteq \mathsf{ref}_f(\rho, B, C)$.
- $\mathsf{ref}_f(\rho) \neq \rho$ *implies that* $\mathsf{ref}_f(\rho, B, C) \neq \rho$ *for some* B *and* C.

In other words, we make no mistakes and we can't get stuck.

Proof.

$\mathsf{ref}_f(\rho)$ is $\bigsqcap_{C,B} \mathsf{ref}_f(\rho, B, C)$ and thus finer than each part.

If $\mathsf{ref}_f(\rho) \neq \rho$ there must be some block B and $p, q \in B$ such that $\neg(f(p)\ \rho\ f(q))$.

Let C be the block containing $f(p)$, done. □

Of course, from a complexity perspective this may not sound too promising: we are breaking one giant step into multiple baby steps. It is not unreasonable to suspect that this might even increase running time.

But: The baby steps provide much better control over the selection of the next refinement step: we can choose the blocks involved at will.

With a little effort this feature can be exploited sufficiently to speed up the whole process.

Suppose we have $\Sigma = \{a\}$ and we want to insure compatibility with δ_a (see below for the general case). In addition, we have an initial partition $(F, Q - F)$. The algorithm maintains two data structures, both are initialized by the given partition.

- a partition P of Q, representing the equivalence relation,
- a split list S with entries some of the blocks in the partition.

We refer to the blocks C in S as active: those are the blocks that we will use at some point to try to refine P by $a^{-1}C$.

The algorithm extracts an active block from the split list and tries to refine the blocks in the partition accordingly.

It then updates the split list in a clever way, and stops when the split list becomes empty.

Initializing P is straightforward, just the given partition.

The split list S only gets the smaller of the two blocks (we are dealing with the single-function case here, see below on how to handle larger alphabets).

Since we are interested in the case where $f = \delta_a$, let us lighten notation a bit and write $a^{-1}C$ rather than $\delta_a^{-1}(C)$.

Incidentally, for some application one needs to allow initial partitions with more than 2 blocks; everything we do here easily generalizes.

Hopcroft's Algorithm

initialize partition P and split list S

while S not empty **do**
    extract C from S
    compute $\widehat{C} = a^{-1}C$
    foreach block B split by C **do**
        $B^+ = B \cap \widehat{C}$
        $B^- = B - B^+$
        replace B by B^+ and B^- in P // update partition
        if B is in S // update split list
        then replace B by B^+, B^- in S
        else replace B by the smaller of B^+, B^- in S
end

Missing Work?

At first glance it may seem like we are not doing enough work: in the last case it feels like both B^+ and B^-, the parts of B obtained by splitting wrto the critical block C, should be added to the split list.

Otherwise we might miss out on splitting some other block that is not split by B but split by B^+ or B^-. The algorithm might stop without having produced an f-compatible relation.

The correctness proof hinges on showing that this cannot happen: in the critical case, if B^+ splits some block, then so does B^-, and the other way around.

By picking the smaller of the two blocks, we get the desired speedup while sill producing the right result.

Hopcroft's Proposition

The following observation by Hopcroft makes this idea of "no-missing-work" more formal. Suppose we have 2 blocks B, C in partition P. Write

$$B/C$$

for the partition induced by splitting B via $a^{-1}C$: $B^+ = B \cap a^{-1}C$ and $B^- = B - a^{-1}C$.

Let X be a block.

Proposition

$$X/B \sqcap X/B^+ = X/B \sqcap X/B^- = X/B^+ \sqcap X/B^-$$

Call a set $Z \subseteq Q$ of states safe for partition P (or simply P-safe) if $f^{-1}(Z)$ does not split any block in P.

We will show that the following assertion is a loop invariant:

$$\forall\, X \in P - S\, \exists\, T \subseteq S \left(X \cup \bigcup T \quad P\text{-safe} \right)$$

In other words, $Z = X$ itself may split some blocks, but if we pad it out with a few blocks in S (and thereby inflate $f^{-1}(Z)$) no splits occur.

This assertion holds before the loop ever executes: there are only two blocks X, Y in P and one, say, Y in S (the smaller of the two). Since $f^{-1}(X \cup Y) = f^{-1}(Q) = Q$, no splits occur.

After the loop terminates with $S = \emptyset$, the claim

$$\forall\, X \in P \left(X\ P\text{-safe}\right)$$

meaning the every block is safe. Done.

As is customary, we indicate the value of a variable after one more execution of the loop-body by attaching a prime: so P' is the partition after one more round.

In the following we argue about the state of affairs at the end of a round. We need to show that

$$\forall\, X \in P - S\, \exists\, T \subseteq S \left(X \cup \bigcup T \quad P\text{-safe} \right)$$

implies that

$$\forall\, X \in P' - S'\, \exists\, T \subseteq S' \left(X \cup \bigcup T \quad P'\text{-safe} \right)$$

So assume we have some arbitrary block $X \in P' - S'$.

Induction arguments about procedural code are typically messy because one has to distinguish between changes in state; in particular variables change their values over and over again.

The resulting arguments can get quite involved combinatorially and are often quite messy.

In this case, X is an old block that has just been removed from the split list.

Since X is old, it has not been split in the last round, yet was removed from the split list.

The only way this can happen is that $C = X$ is the critical block, and was removed after splitting blocks wrto $a^{-1}C$. But then safety is a direct result of the construction.

By induction hypothesis: $Z = X \cup \bigcup T$ is P-safe for some $T \subseteq S$.

Case 2.1: $C \notin T$

In this case we can replace split blocks in the padding set T: replace Y by Y^+ and Y^- (which are both active) and we have $Z = X \cup \bigcup T'$ is P-safe. It is easy to check that Z is also P'-safe.

Case 2.2: $C \in T$

Again, we can replace split blocks in the padding set T, producing T'. However, we can no longer use the critical block $C \notin S'$, we can only pad with $T'' = T' - C$. But no block in P' can be affected by this: it was already split wrto $f^{-1}(C)$ during the round.

Case 3: $X \in P' - P$

So block X is new and was created by splitting a block Z in the last round.

Say, $X = Z^+$ and $Z \cup \bigcup T$ is P-safe for some $T \subseteq S$.

As before we can handle split blocks in T.

Case 3.1: $C \notin T$

Then $X \cup \bigcup T' \cup Z^-$ is P'-safe as in Case 2.1. Note that indeed $Z^- \in S'$ by construction.

Case 3.2: $C \in T$

In this case, $X \cup \bigcup T_0 \cup Z^-$ is P'-safe as in Case 2.2.

Each block in P is represented by a doubly-linked list.

We maintain an array of pointers to these lists for P and similarly for S. We also keep track of the cardinality of each block.

Furthermore, we have an array of pointers so that pos[p] points to the list node containing p, plus information about the current block containing p.

The key part of each round is the computation of $\widehat{C} = a^{-1}C$. We may assume that $f^{-1}(p)$ has been precomputed for each state p. We can traverse C in time linear in $|C|$.

When a block B is hit for the first time, we start splitting it into two lists B^+ and B^-. If, in the end, $B^- = \emptyset$ we simply replace B by B^+.

All this can be handled in time $O(|a^{-1}C|)$.

Active States

Let us say that a state p is active if $p \in \bigcup S$, inactive otherwise.

At level 0, at most half of all states are active.

Each state in the critical block C becomes inactive, but maybe reactivated later. Hence we can naturally assign activation levels 0, 1, 2, ... to all active states.

Recall that we only reactivate at most half the states in a block, so no state can be activated more than $\log n$ times. But then the total work computing pre-images of active states during the whole execution is just $O(n \log n)$.

Hence the running time of the whole algorithm is bounded by $O(n \log n)$.

J. Hopcroft

A N log N Algorithm for Minimizing States in a Finite Automaton

STAN-CS-71-190

This is a seminal paper that will bring tears to your eyes.

We modify the split list in the algorithm to contain entries

$$(a, C) \in S$$

where C is a block and $a \in \Sigma$: the intent is that we later refine via $a^{-1}C$.

Of course, (a, X) is smaller than (a, Y) if $|X| \leq |Y|$.

Note that we need to add (a, X) for all $a \in \Sigma$, which produces a running time of $O(kn \log n)$.

Hopcroft Example

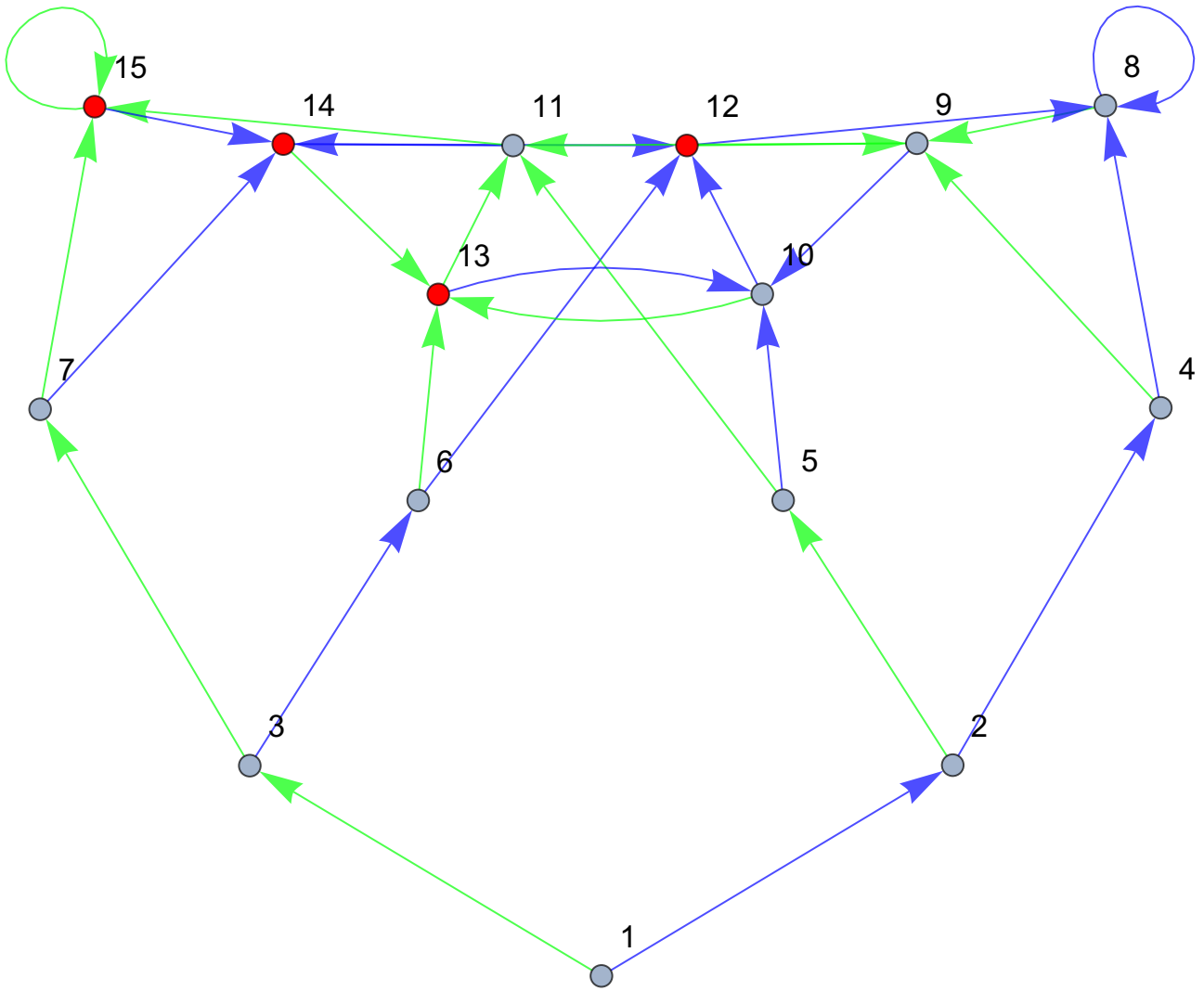
The following example uses a machine over alphabet $\{a, b\}$ with 15 states. The transition matrix is

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	2	4	6	8	10	12	14	8	10	12	14	8	10	12	14
b	3	5	7	9	11	13	15	9	11	13	15	9	11	13	15

The final states are $\{12, 13, 14, 15\}$.

The following table shows the element extracted from the split list in the first column and the blocks in the second.

Note that split list entry (a, i) means: use the ith block in the current partition with respect to $f = \delta_a$.


15
14
11
12
9
8
13
10
7
4
6
5
3
2
1

$((12,13,14,15),(1,2,3,4,5,6,7,8,9,10,11))$	$a,1;b,1$
$((14,15),(1,2,3,4,5,6,7,8,9,10,11),(12,13))$	$b,1;a,3;b,3$
$((14,15),(6,7,10,11),(12,13),(1,2,3,4,5,8,9))$	$b,1;a,3;b,3;a,2;b,2$
$((15),(6,7,10,11),(12,13),(1,2,3,4,5,8,9),(14))$	$a,3;b,3;a,2;b,2;a,5;b,5$
$((15),(7,11),(12,13),(1,2,3,4,5,8,9),(14),(6,10))$	$a,3;b,3;a,2;b,2;a,5;b,5;a,6;b,6$
$((15),(7,11),(13),(1,2,3,4,5,8,9),(14),(6,10),(12))$	$a,5;b,5;a,6;b,6;a,7;b,7$
$((15),(7,11),(13),(3,5,9),(14),(6,10),(12),(1,2,4,8))$	$a,5;b,5;a,6;b,6;a,7;b,7;a,4;b,4$

In the first step, $a^{-1}(12,13,14,15) = (6,7,10,11,14,15)$ and we split both existing blocks.

This trace only shows steps where the partition changes. Note there are many "useless" steps at the end.

The algorithm is quite messy to implement correctly, as can be seen from the following papers:

D. Gries
Describing an algorithm by Hopcroft
Acta Informatica, 2 (1973) 97–109.

T. Knuutila
Re-describing an algorithm by Hopcroft
Theoretical Computer Science, 250 (2001) 333–363.

Exercise

Implement Hopcroft's algorithm, correctly.

Theorem (Hopcroft 1971)

Hopcroft's algorithm minimizes a DFA in $O(kn \log n)$ steps, where n is the state complexity of the DFA and k the size of the alphabet.

Gries gives a very careful description (and slight improvement) of the algorithm, really proves correctness and analyzes running time in detail.

Knuutila pointed out in 2001, one can produce cubic (in n) running time when $k = n/2$ and a poor method of choosing the "smaller" block is used. OK, but how is this relevant?

Note that Hopcroft's algorithm is nondeterministic in several ways.

- We can extract any element from the split list (e.g., could use a stack, queue, ...).
- Likewise we can place the new entries anywhere in the split list.
- When B^+ and B^- have the same size, we can pick either one.

None of these choices effect correctness, but they may well influence running time. As a consequence, any detailed analysis taking into account possible strategies is quite complicated.

It should be noted that the algorithm often takes far fewer than $n \log n$ steps.

Given a reasonable implementation, every round is linear. It turns out to be quite difficult to construct inputs where the algorithm requires $\log n$ many rounds.

The best result known today is that there are some DFAs such that the algorithm takes $n \log n$ steps for a certain choice of active blocks in the main loop.

Alas, for these machines a different choice of active blocks results in linear running time.

- When is the running time $\Omega(n \log n)$ regardless of the chosen split list protocol? (A unary example is known where the execution sequence is essentially unique and reaches the log-lin bound.)

- What is the average complexity of Hopcroft's algorithm (average with respect to input automaton and/or split list protocol)?

- Is Hopcroft faster than Moore on average? For the uniform distribution, Moore has expected behavior $O(n \log n)$ and it may be that the constants are smaller (Bassino, David, Nicaud 2009).

There are many examples where the number of transitions m in a partial DFA is much smaller than $k \cdot n$, k the size of the alphabet and n the number of states. This leads naturally to an algorithmic question:

> Is there a $O(m \log n)$ minimization algorithm that deals directly with partial transition functions?

This would also nicely encapsulate problems with alphabet size in a parameter that really reflects the size of the data structure: unused symbols do not inflate the transition function.

The high-level logic is similar as in Hopcroft's algorithm: one maintains a partition of Q and tries to refine the partition by splitting with sets of the form

$$\delta_a^{-1}(B)$$

The choice of B is a bit more complicated, though. Special care is taken to avoid unnecessary computation when preimages of blocks are computed.

New Idea:
Maintain and refine a second partition of the transitions.

The transition partition helps to speed up pre-image computation.

Important implementation detail: the partition data structure is all array-based (unlike Hopcroft's algorithm).

Suppose we wish to maintain a partition of $\underline{n}$[†]. Keep track of r, the number of blocks, and maintain two maps (arrays)

$$\mathsf{lo}, \mathsf{hi} : \underline{r} \longrightarrow \underline{n}$$

plus an array `P[n]` such that

> B_d, the block number d, is located in $P[\mathsf{lo}[d], \mathsf{hi}[d] - 1]$

We also have location and block-number maps

$$\mathsf{loc} : \underline{n} \longrightarrow \underline{n}$$
$$\mathsf{bnum} : \underline{n} \longrightarrow \underline{r}$$

such that $P[\mathsf{loc}[p]] = p$, $\mathsf{loc}[P[i]] = i$ and $p \in B_{\mathsf{bnum}[p]}$.

[†]Write $\underline{n}$ for $\{0, 1, \ldots, n-1\}$.

It is convenient to subdivide the splitting process into three phases:

- Pre-splitting: initialize offset pointers $\mathsf{mrk}[d] = \mathsf{lo}[d]$ for all blocks, create empty hit list.
- Splitting: process a sequence of elements, swap each to the "marked" part $P[\mathsf{lo}[d], \mathsf{mrk}[d] - 1]$ of their respective blocks. If block B_d is encountered for the first time, add to hit list.
- Post-splitting: walk through blocks in hit list and update to maintain invariants.

It is straightforward to arrange the post-splitting phase so that whenever B splits into B_1 and B_2, the larger part replaces B and the smaller part receives the higher block index (the current value of r).

As already mentioned, the minimization algorithm uses two PRDS:

- P: represents a partition of Q; initialized to $(F, Q - F)$, as usual.
- T: represents a partition of the transitions; initialized to blocks containing all transitions with the same label.

The main loop of the algorithm looks like this:

foreach T-block C **do**
    split blocks in P via C
    foreach P-block B **do**
        split blocks in T via B

New blocks are "appended" in both partitions, so the traversals end when all blocks have been processed.

pre-split P
foreach transition $p \xrightarrow{a} q$ in block C **do**
    mark source p in P-partition
post-split P

pre-split T
foreach state p in block B **do**
    foreach transition $t : q \xrightarrow{a} p$ **do**
        mark t in T-partition
post-split T

Note that for the transition splitting operation one needs to be able to traverse all transitions with a fixed target.

To this end one precomputes two arrays

$$\mathsf{trn} : \underline{m} \longrightarrow \underline{m}$$
$$\mathsf{fst} : \underline{n} \longrightarrow \underline{m}$$

such that for each state p

> the transitions with target p are located in $\mathsf{trn}[\mathsf{fst}[p], \mathsf{fst}[p+1]-1]$

This is easy via counting sort.

Recall that $\mathsf{bnum}(p)$ is the block number of state p. By abuse of notation, $\mathsf{bnum}(t)$ is the block number of transition t.

Proposition

- *States:*
  $\mathsf{bnum}(p) \neq \mathsf{bnum}(q)$ *implies* $[\![p]\!] \neq [\![q]\!]$
- *Transitions:*
  $\mathsf{bnum}(s) \neq \mathsf{bnum}(t)$ *and* $\mathsf{lab}(s) = \mathsf{lab}(t)$ *implies* $[\![\mathsf{trg}(s)]\!] \neq [\![\mathsf{trg}(t)]\!]$

Proposition

Let s, t be two transitions in the same T-block.
If src(s) *and* src(t) *are in different P-blocks, then at least one of the blocks is unprocessed.*

Proposition

Let p, q be two states in the same P-block and $s : p \xrightarrow{a} p'$, $t : q \xrightarrow{a} q'$ two transitions in different T-blocks. Then at least one of these blocks is unprocessed.
If $s : p \xrightarrow{a} p'$ and there is no a-transition with source q, then s is in an unprocessed T-block.

Theorem

The algorithm correctly minimizes a trim partial DFA in $O(m \log n)$ steps.

Proof.

It follows from the propositions that two states have the same behavior iff, upon completion, they are in the same P-block.

For running time, note that since we are dealing with a trim automaton, we have $n \leq m + 1$.

As in Hopcroft's algorithm, each state can be active at most $\log n$ times and, likewise, a transition can be active at most $\log m$ times.

So the total running time is $O(m \log n)$.

Antti Valmari

Fast brief practical DFA minimization

Information Processing Letters 112 (2012) 213–217

This is another paper that will bring tears to your eyes, but for entirely different reasons.