CDM

Minimization Algorithms

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- Implementing Equivalence Relations
- State Merging: Forward
- Equivalence Testing
- State Merging: Backward (aka Hopcroft's Algorithm)

Computing (Behavioral) Equivalence

Recall: Covers

Recall from last time that there is a close connection between an arbitrary DFA and the corresponding minimal DFA.

Lemma 1. *Let L be a regular language and M an arbitrary accessible DFA for L. Then there is cover from M onto M_L.*

The covering map is given by the behavior of a state:

$$f : Q \to Q_L$$
$$f(p) = [[p]]$$

Minimization Algorithms

The covering map also provides a way to minimize a DFA M: all we need to do is to merge all the states that map to the same quotient: behavioral equivalence is the kernel relation defined by the cover map.

But note that there is a bit of a vicious cycle: to compute the cover f directly we need M_L. If we have the latter there is no need to minimize M.

Nonetheless, covers indicate the right approach to efficient algorithms:

- Start with any DFA M for L.
- Remove inaccessible states from M.
- Compute the behavioral equivalence relation for M.
- Lastly, merge states with the same behavior.

Implementing Equivalence Relations

Formally, an equivalence relation on A is a set $E \subseteq A \times A$ with certain properties (reflexive, symmetric, transitive).

Suppose $|A| = n$ and $|E| = m$, so $n \leq m \leq n^2$.

data structure	test	space
list of pairs	$O(m)$	$\Theta(m)$
sorted list of pairs	$O(\log n)$	$\Theta(m)$
boolean matrix	$O(1)$	$\Theta(n^2)$
selector function	$O(1)$	$\Theta(n)$
union/find	$\approx O(1)$	$\Theta(n)$

Only the last two representations are of interest if we are looking for fast algorithms.

Total Recall, I

Definition 1. *Given a map* $f : A \longrightarrow B$ *the* kernel relation *induced by* f *is the equivalence relation*

$$x \; K_f \; y \iff f(x) = f(y).$$

Note that K_f is indeed an equivalence relation.

This may seem somewhat overly constrained, but in fact every equivalence relation is a kernel relation for some appropriate function f by GANS. We can even pick $f : A \to A$.

To test whether $a, b \in A$ are equivalent we only have to compute $f(a)$ and $f(b)$ and test for equality.

If the values of f are stored in a table this is $O(1)$, with very small constants.

The Canonical Selector Function

We may safely assume that A carries some natural total order.

In fact, usually $A = [n]$ and we can store f as a simple array: this requires only $O(n)$ space and equivalence testing is $O(1)$ with very small constants.

Definition 2. *The* canonical selector function *for an equivalence relation R on A is*

$$\mathsf{sel}_R(x) = \min\big(\, z \in A \mid x \;\rho\; z \,\big)$$

So each equivalence class is represented by its least element.

This is really quite natural. E.g. congruence classes modulo 3 on $\mathbb{N}$ are represented by 0, 1, 2.

Example

Consider the equivalence relation E on $[10]$ with blocks

$$(1, 3, 5, 7, 9), (2, 6, 10), (4), (8)$$

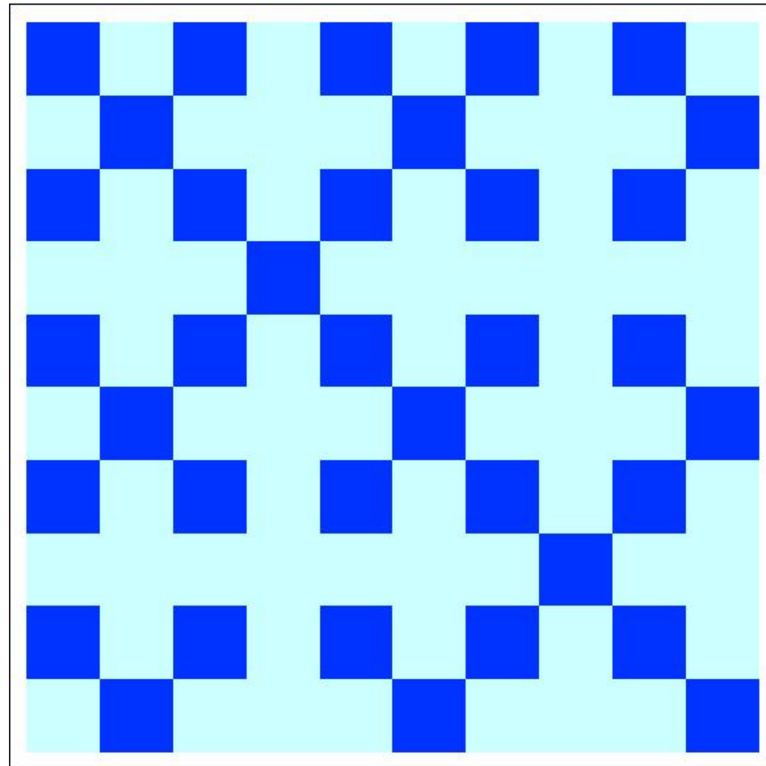
List of pairs

$$(1, 1), (1, 3), (1, 9), \ldots, (9, 1), (2, 2), (2, 6), \ldots, (10, 2), (4, 4), (8, 8)$$

Sorted list of pairs: well, just sort the thing . . .

Example, contd.

Boolean matrix



Example, contd.

Kernel representation

Note that $p \; E \; q \iff \nu_2(p) = \nu_2(q)$ where $\nu_2(x) = \max\big(\, k \mid 2^k \text{ divides } x \, \big)$.

Hence E is the kernel relation of ν_2.

The canonical selector function is

$$(1, 2, 1, 4, 1, 2, 1, 8, 1, 2)$$

Total Recall, II

Definition 3. *Let E be an equivalence relation on A and $B \subseteq A$. Then E* saturates *B if B is the union of equivalence classes of E.*

In other words,

$$B = \bigcup_{x \in B} [x]_E.$$

Proposition 1. *In any DFA, the behavioral equivalence relation saturates the set of final states.*

This means that behavioral equivalence is a refinement of the basic partition $(F, Q - F)$.

We can use this as the starting point in an approximation algorithm.

Total Recall, III

We also need some simple manipulations of equivalence relations.

Definition 4. Meet *of Equivalence Relations*
Let ρ and σ be two equivalence relations on A. Then $\rho \sqcap \sigma$ denotes the coarsest equivalence relation finer than both ρ and σ.

In other words,

$$x \ (\rho \sqcap \sigma) \ y \iff x \ \rho \ y \land x \ \sigma \ y.$$

This is sometimes written $\rho \cap \sigma$ which is fine when the relation is represented as a set of pairs, but a bit misleading otherwise.

Also note that for the dual notion of join of ρ and σ things are much more complicated.

Exercise 1. *Figure out how to compute the join $\rho \sqcup \sigma$.*

Attempt 1: General Abstract Nonsense

In principle, all we need to do is to compute the equivalence relation associated with the behavioral map: we already know that, given any accessible DFA $M = \langle\, Q, \Sigma, \delta, q_0, F \,\rangle$, the behavioral equivalence relation E is the kernel relation of the behavior map:

$$[\![.]\!] : Q \longrightarrow \mathfrak{P}(\Sigma^\star), \qquad p \mapsto [\![p]\!]$$

This an elegant characterization, but way too abstract; it has no direct computational meaning.

Of course, we could try to pull our standard trick to represent the quotients by DFAs, but that would yield no improvement over the algorithm from last class. We want something really efficient at this point.

Attempt 2: Approximation Algorithm

The main idea is to start with the very rough approximation $(F, Q - F)$ and then refine this equivalence relation till we get behavioral equivalence.

Assume without loss of generality that M has state set $[n]$. To represent the equivalence relations that appear during the computation we use selector functions

$$f : [n] \longrightarrow [n]$$

Needless to say, the approximation algorithm could use any other representation (such as lists of pairs or Boolean matrices) just as well, but selector functions yield very good running time. The code becomes much easier, too.

Forward Algorithm

At level 0 we approximate E by $E_0 = (Q - F, F)$.

So E_0 is coarser that E and we need to refine (subdivide the blocks into smaller ones) it a bit.

We will obtain a sequence of better and better approximations $E_0, E_1, \ldots, E_k$ where $E_k = E$.

Nice plan, but how do we compute E_{k+1} from E_k? What are the possible obstructions to E_k being E?

Key Insight: The only possible obstruction to E_k being behavioral equivalence is that for some $p, q \in Q$:

$$p \; E_k \; q \quad \text{but not} \quad \delta(p,a) \; E_k \; \delta(q,a)$$

Forward Algorithm, Contd.

Let's take a closer look. For simplicity write $\delta_a : Q \to Q$, $\delta_a(p) = \delta(p, a)$.

- If the functions δ_a, $a \in \Sigma$, map all blocks in our approximation E_k into other blocks we are done.
- Otherwise, if δ_a does not stay within blocks we have to refine our approximation E_k.

Even better, in the second case the mismatch tells us how to split up the blocks of E_k.

More precisely, for any equivalence relation ρ on Q define

$$p \; E^a \; q \iff \delta_a(p) \; E \; \delta_a(p)$$

Forward Algorithm, Contd.

If $E^a = E$ we do nothing, otherwise compute the meet of the two equivalence relation.

Of course, we need to do this for all letters of the alphabet.

So, algebraically, we refine E_k to

$$E_{k+1} = E_k \sqcap E_k^{a_1} \sqcap E_k^{a_2} \sqcap \ldots \sqcap E_k^{a_n}$$

where $\Sigma = \{a_1, \ldots, a_n\}$.

Termination is easy: Stop when for the first time $E_k = E_{k+1}$.

Exercise 2. *Show how to streamline this algorithm by running through a sequence of small refinement steps $E \mapsto E^a$ as a runs through Σ.*

State Merging Algorithm

Once we have computed the behavioral equivalence relation E (or, for that matter, any other compatible equivalence relation on Q) we can determine the quotient structure.

Replace Q by Q/E; q_0 and F by the corresponding equivalence classes.

Define

$$\delta'([p]_E, a) = [\delta(p, a)]_E$$

Proposition 2. *This produces a new DFA that is equivalent to the old one, and reduced.*

Exercise 3. *Show that this merging really produces a DFA (rather than some random finite state machine.*

So far, so good . . .

We can represent any equivalence relation on $[n]$ easily in $\Theta(n)$ space, with equivalence testing $O(1)$.

But we are in a dynamic situation: we have to compute new equivalence relations from given ones.

More precisely, we are given a relation R represented by the canonical selector sel_R. We need to compute the canonical selector sel_T for the meet

$$T = R \sqcap R^a$$

And, of course, we want to do this cheaply.

Meet and Selectors

It is easy to compute the canonical selector for R^a, we only need to perform functional composition:

$$S(p) = \mathsf{sel}_R(\delta_a(p)).$$

In code, this is just a double table lookup and lighning fast. Since

$$p \; T \; q \iff \mathsf{sel}_S(p) = \mathsf{sel}_S(q) \wedge \mathsf{sel}_R(p) = \mathsf{sel}_R(q)$$

we are looking for identical pairs in the columns of the table

1	2	3	$\ldots$	p	$\ldots$	n
$S(1)$	$S(2)$	$S(3)$	$\ldots$	$S(p)$	$\ldots$	$S(n)$
$\mathsf{sel}_R(1)$	$\mathsf{sel}_R(2)$	$\mathsf{sel}_R(3)$	$\ldots$	$\mathsf{sel}_R(p)$	$\ldots$	$\mathsf{sel}_R(n)$

Meet Algorithm

Traverse the table left to right.

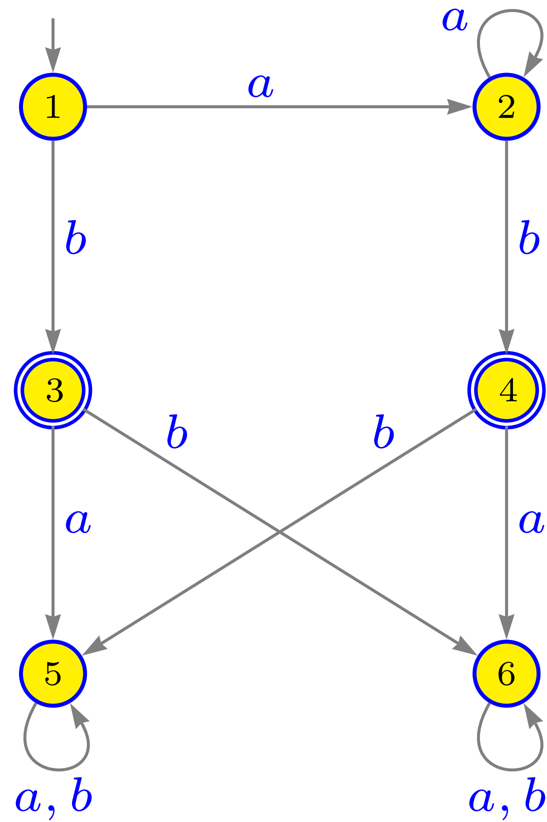
Every time a new pair (i, j) in the S/sel_R rows appears, set $T(p) = p$.

Otherwise set $T(p) = q$ where q is minimal with the same S/sel_R pair.

```
//  meet( R, S )
for( p = 1 .. n ) {
        i = s[p];                    // selector for R^a
        j = sel_r[p];                // selector for R
        if( (i,j) is new  )
                t[p] = val(i,j) = p;
        else
                t[p] = val(i,j);
}
```

Merging Example

Recall the machine for $L = a^{\star}b$.



Computing Behavioral Equivalence

The transition matrix is

	1	2	3	4	5	6
a	2	2	5	6	5	6
b	3	4	6	5	5	6

with final states $\{3,4\}$:

	1	2	3	4	5	6
E_0	1	1	3	3	1	1
a	1	1	1	1	1	1
b	3	3	1	1	1	1
E_1	1	1	3	3	5	5
a	1	1	5	5	5	5
b	3	3	5	5	5	5
E_2	1	1	3	3	5	5

Hence $E_2 = E_1$ and the algorithm terminates. Merged states are $\{1,2\}$, $\{3,4\}$, $\{5,6\}$.

To save space, we have performed refinement steps $E \mapsto E \sqcap E^a \sqcap E^b$. In a real implementation this would be broken up into two steps.

Another Example

Consider the DFA with final states $\{1, 4\}$ and transition table

	1	2	3	4	5	6	7	8
a	2	4	5	2	6	8	4	6
b	3	5	4	3	7	4	8	7

produces the trace:

	1	2	3	4	5	6	7	8
E_0	1	2	2	1	2	2	2	2
a	2	1	2	2	2	2	1	2
b	2	2	1	2	2	1	2	2
E_1	1	2	3	1	5	3	2	5
a	2	1	5	2	3	5	1	3
b	3	5	1	3	2	1	5	2
E_2	1	2	3	1	5	3	2	5

Number of Stages

In both examples above $E_1 = E_2$ and the construction ends after two rounds.

Of course, this is pure coincidence (actually not: otherwise the trace won't fit on one slide).

Exercise 4. *Perform the forward state merging algorithm on some reasonably small DFAs by hand. Then implement the algorithm.*

Exercise 5. *Find a DFA on n states where the merging process takes $n-1$ stages. Note that $n-1$ is an upper bound, so things could not be any worse.*

Running Time, cont.

How hard is it to compute E_{k+1} from E_k?

Using the canonical selector function to represent the equivalence relations this step is expected linear time using a hash table.

Proposition 3. *The total expected running time for minimization is $O(kn^2)$ where k is the size of the alphabet, and n the size of the given DFA.*

Exercise 6. *Figure out how to guarantee linear time for each stage at the cost of a quadratic time initialization. Hint: use a $n \times n$ array of integers initialized to -1; enter the (i, j) pairs into the array and reset to -1 after each round. Discuss advantages and disadvantages of this method.*

Minimization

Theorem 1. *Given a DFA for a regular language L one can compute the minimal DFA for L in quadratic time.*

This may seem underwhelming compared to our brute-forcish algorithm from last class but note

- The size of the alphabet enters linearly here: $O(kn^2)$, so the constants are better.
- Often less than n rounds are needed, so we get performance $O(knn')$ for some $n' \leq n$.
- The space requirement is linear.
- The code is amazingly simple.

Equivalence Testing

Isomorphism Testing

Note that one can produce variants of a DFA by permuting the states. Clearly, these variants are all equivalent, but how do we check? More precisely, two DFAs M_1 and M_2 are *isomorphic* if there is a bijection $f : Q_1 \longrightarrow Q_2$ such that

$$f(q_{10}) = q_{20}$$

$$f(\delta_1(p, a)) = \delta_2(f(p), a)$$

$$f(F_1) = F_2$$

Thus, in isomorphic DFAs the states are just renamed.

As usual, there is an associated decision problem.

Problem: **Isomorphism of DFA**
Instance: Two DFAs M_1 and M_2.
Question: Are M_1 and M_2 isomorphic?

Isomorphism Testing

To check whether two DFAs are isomorphic one can use a variant of depth-first-search. We may safely assume the machines have the same size.

- Set $f(q_{01}) = q_{02}$.
- Extend the domain of f according to δ_1, and the range according to δ_2.
- Stop with failure if there ever is a clash (new point on one side, old point on the other; two different old points).
- Check if f maps the final states properly. If so, return Yes, otherwise return No.

Exercise 7. *Figure out the details of this isomorphism testing algorithm.*

Equivalence Testing

Theorem 2. *Given two DFAs one can test whether they are equivalent in quadratic time.*

Proof.

Given M_1 and M_2 we can compute the corresponding minimal automata M_1' and M_2' in quadratic time. Then we can check in linear time whether M_1' and M_2' are isomorphic. □

Again, this may not seem much of an improvement compared to the sledgehammer algorithm from last time. But it is, for reasons similar to the ones listed above.

Better Equivalence Testing

The isomorphism testing part of our algorithm is very fast, but minimization is expensive.

Wild Idea: Can we somehow avoid minimization?

Suppose the two given DFAs have disjoint state sets Q_1 and Q_2 and set $Q = Q_1 \cup Q_2$. We will define an equivalence relation E on Q as follows.

```
set E = identity;
set active = ( (q01,q02) );

while( active != empty ) {
        (p,q) = active.pop();
        if( ! p E q ) {
                set p E q;
                forall( a in S ) do
                        add  (delta(p,a),delta(q,a)) to active;
        }
}
```

Algorithm Details

The algorithm returns false whenever two states $p \in Q_1$ and $q \in Q_2$ are defined to be equivalent under E but

$$p \in F_1 \text{ xor } q \in F_2.$$

Otherwise we return true.

We can use a simple stack for the active list, the interesting question is how to maintain the equivalence relation E.

Note that this is the dynamical situation: initially E is just the identity relation, but as we go along we discover more related pairs.

So really we need to compute the *equivalential closure* of all the pairs (p, q) we have discovered.

Union/Find

Instead of using the (static) canonical selector function we use a slightly more general representation.

We use a collection of trees, initially all consisting of a single node.

Each tree represents an equivalence class and is directed towards the root.

When a new pair (p, q) is encountered we proceed as follows:

- Find the roots r and r' of the trees containing p and q, respectively.
- If $r = r'$ we do nothing.
- Otherwise, we merge the two trees.

Note that the running time will depend on the depth of the trees: we have to march all the way to the root.

Implementation Details

All we need to represent the collection of trees is a simple array T where $T[p]$ is the predecessor of p in the tree p belongs to.

For any root r we have $T[r] = r$.

To find the root of the tree of p we do

```
findroot( p ) {
        if( p != T[p] )
                return findroot( T[p] );
        else
                return p;
}
```

To merge two trees with roots r and r' we simply set $T[r] = r'$. Or $T[r'] = r$.

Union/Find Heuristics

There are two improvements to make this algorithm enormously efficient.

Ranked union: attach the shallower tree to the deeper one.

Path compression: attach all elements in the branches to p and q directly to the new root.

Theorem 3. *The Union/Find algorithm has essentially linear running time.*

The running time is $O(m\,\alpha(n))$ where m is the number of operations and n the size of the carrier set. Here α is the "inverse" of the Ackermann function:

$$\alpha(n) = \min\big(\, k \mid n \leq A(k,k) \,\big)$$

So, in theory $\alpha(n)$ tends to infinity – but in the real world $\alpha(n)$ is a small constant.

Algorithm Explained

By initialization $q_{01} \; E \; q_{02}$.

If $p \in Q_1$ and $q \in Q_2$ such that $p \; E \; q$ then $\delta(p, a) \; E \; \delta(q, a)$ for each $a \in \Sigma$.

By induction for any word x

$$\delta(q_{01}, x) \; E \; \delta(q_{02}, x).$$

Using a loop invariant one can show that in fact

$$E \sqcap Q_1 \times Q_2 = \{ \, (\delta(q_{01}, x), \delta(q_{02}, x)) \mid x \in \Sigma^\star \, \}$$

Exercise 8. *Give a detailed proof for the correctness of this algorithm.*

Ponder Deeply

There is a $O(n \log n)$ minimization algorithm that's a bit more difficult to explain, and the running time analysis is harder (see below). However, no better algorithm is known to date.

Exercise 9. *Why can't this algorithm (or rather: some slight modification thereof) be used to compute the minimal automaton in nearly linear time?*

Note that we do not claim that equivalence is polynomial time decidable for nondeterministic machines. In fact, one can show that equivalence testing is $\mathbb{NP}$-hard (even PSPACE-hard) for nondeterministic machines, even if one of the two machines simply accepts all words over the input alphabet.

Likewise it is computationally hard to find minimal nondeterministic machines for a given language. Moreover, the minimal nondeterministic machine is not unique in general.

A Fast Minimization Algorithm

Hopcroft's Algorithm: Going Backward

Like the standard algorithm, Hopcroft's algorithm uses stepwise refinement of the original final-nonfinal partition.

However, instead of looking at the images of states under δ_a, we consider pre-images.

Consider some map $f : Q \to Q$, in the actual algorithm these will be the δ_a's.

Suppose π is a partition of Q, and consider a subset $B \subseteq Q$.

We say that B *splits* π if for some block X of π:

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

In other words, both $f(X)$ and $f(Q - X)$ intersect B, so if we were to

Splitting and Refining

Note that a partition all of whose blocks are non-splitting must be f-compatible: if x and y are equivalent, so are $f(x)$ and $f(y)$.

If B splits π we fix this problem by refining the partition. To this end define a new two-block equivalence relation

$$\pi(f, B) = (f^{-1}(B), Q - f^{-1}(B))$$

and refine:

$$\pi \mapsto \pi \sqcap \pi(\delta_a, B)$$

Giants versus Babies

One (giant) step in the standard algorithm is essentially of the form

$$\pi \mapsto \pi \sqcap \pi(f)$$

where $x \; \pi(f) \; y \iff f(x) \; \pi \; f(y)$ and $f = \delta_a$, $a \in \Sigma$.

By contrast, Hopcroft's algorithm focuses on a single block of $\pi(f)$ in a single (baby) step. In the end we get the same result since

$$\pi(f) = \bigwedge_{B \text{ in } \pi} \pi(f, B)$$

so the forward method is really very similar.

But: the baby steps provide better control over the selection of the next refinement step (we can choose the block at will) and can be exploited to speed up the whole process.

Hopcroft's Algorithm

Here is pseudo-code for the algorithm. For simplicity, we only deal with a single $f = \delta_a$, the real algorithm has to cope with all such functions.

activate the smaller of $Q - F$, F
while there is an active block B
  compute $C = f^{-1}(B)$
  inactivate B
  foreach block D split by B **do**
    compute $D^+ = D \cap C$
    compute $D^- = D - C$
    remove D, add D^+ and D^-
    if D was active
    then mark both D^+, D^- active
    else mark smaller of D^+, D^- active

Note that the algorithm is nondeterministic in the sense that we can pick any active block in the main loop.

Sample Run

	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

$$
\begin{array}{ll}
 & ((12,13,14,15),(1,2,3,4,5,6,7,8,9,10,11)) \\
(a,1) & ((14,15),(1,2,3,4,5,6,7,8,9,10,11),(12,13)) \\
(a,1) & ((14,15),(7,11),(12,13),(1,2,3,4,5,6,8,9,10)) \\
(a,1) & ((15),(7,11),(12,13),(1,2,3,4,5,6,8,9,10),(14)) \\
(a,4) & ((15),(7,11),(12,13),(1,2,3,4,5,6,8,9,10),(14)) \\
(a,4) & ((15),(7,11),(12,13),(1,2,3,4,5,8,9),(14),(6,10)) \\
(a,5) & ((15),(7,11),(12,13),(1,2,3,4,5,8,9),(14),(6,10)) \\
(a,5) & ((15),(7,11),(12,13),(1,2,3,4,5,8,9),(14),(6,10)) \\
(a,6) & ((15),(7,11),(13),(1,2,3,4,5,8,9),(14),(6,10),(12)) \\
(a,6) & ((15),(7,11),(13),(3,5,9),(14),(6,10),(12),(1,2,4,8))
\end{array}
$$

Nothing changes beyond this point but there are several more steps since all of $(a,7)$, $(a,8)$, $(b,1)$, $(b,3)$, $(b,2)$, $(b,5)$, $(b,6)$, $(b,7)$, $(b,8)$ are still active.

Running Time

Theorem 4. *Hopcroft's algorithm minimizes a DFA in $O(k \cdot n \log n)$ steps where n is the state complexity of the DFA and k the size of the alphabet.*

It should be noted that the algorithm often takes far fewer than $n \log n$ steps. In fact, it is quite difficult to construct inputs where it requires this many steps.

The best result know 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.

Research Problems:

Are there any instances that require $\Omega(n \log n)$ steps regardless of which block is chosen?

What is the average complexity of Hopcroft's algorithm?

Summary

- In the class of DFAs every regular language has a natural normal form, the minimal DFA (which can be characterized abstractly in terms of the quotient automaton).
- Quotient constructions are sometimes helpful in building certain finite state machines.
- Because of the uniqueness of the normal form, one can test whether two finite state machines are equivalent.
- Computationally one constructs the minimal DFA by state merging. This is easily accomplished in quadratic time, and, with more effort, in time $n \log n$.
- Equivalence testing can be accomplished in essentially linear time.