Fast Coalgebraic Bisimilarity Minimization

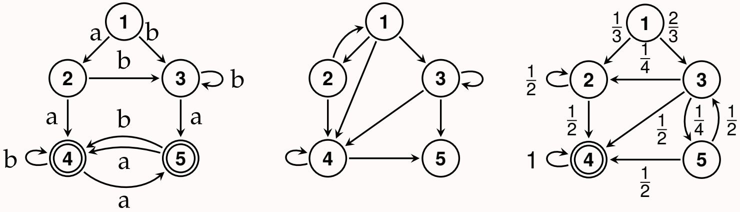
(POPL'23)

Jules Jacobs
Radboud University

Thorsten Wißmann
Radboud University
$\rightarrow$
Friedrich-Alexander-Universität
Erlangen-Nürnberg

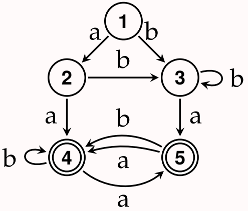
The Automaton Zoo

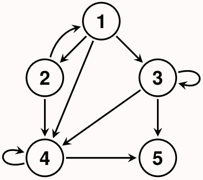
Deterministic finite automata, tree automata, (labeled) transition systems, weighted and probabilistic automata, Markov decision processes, ...

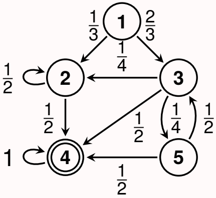


The Automaton Zoo

Deterministic finite automata, tree automata, (labeled) transition systems, weighted and probabilistic automata, Markov decision processes, ...





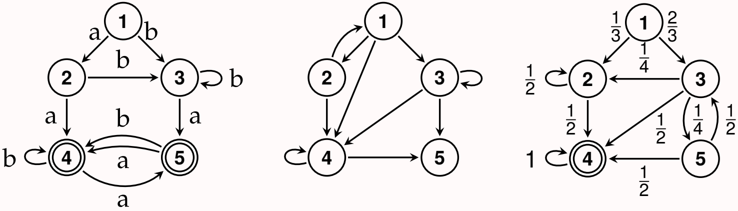


Automaton Minimization

Find and merge behaviorally equivalent states

The Automaton Zoo

Deterministic finite automata, tree automata, (labeled) transition systems, weighted and probabilistic automata, Markov decision processes, ...



Automaton Minimization

Find and merge behaviorally equivalent states

Coalgebraic Bisimilarity Minimization

Algorithms that work for a general class of F-automata

Our contribution

a **fast** and **general** algorithm for minimizing automata

Our contribution

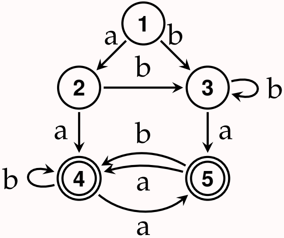
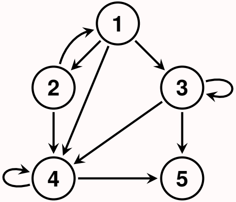
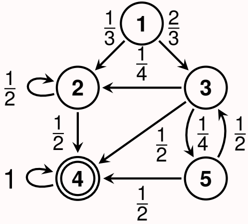
a **fast** and **general** algorithm for minimizing automata

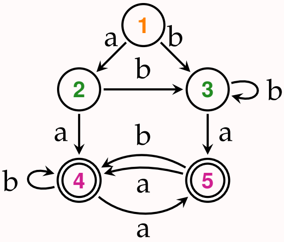
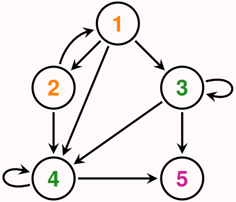
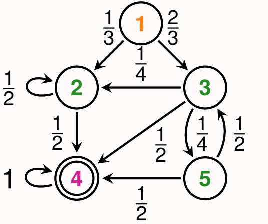
- *General:* works for any computable coalgebra
- *Decent asymptotic complexity:* $\mathcal{O}(\phi_F \cdot m \log n)$
- *Fast in practice:* no penalty for generality
- *Low memory usage:* important for large automata

Examples of Coalgebraic Automata

Automaton type	**Equivalence**	**Functor** $F(X)$
DFA	Language Equivalence	$2 \times A^X$
Transition Systems	Strong Bisimilarity	$\mathcal{P}(X)$
LTS	Strong Bisimilarity	$\mathcal{P}(A \times X)$
Weighted Systems	Weighted Bisimilarity	$M^{(X)}$
Markov Chain	Probabilistic Bisimilarity	$A \times \mathcal{D}(X)$
MDP	Probabilistic Bisimilarity	$\mathcal{P}(\mathcal{D}(X))$
Weighted Tree Automata	Backwards Bisimilarity	$M^{(\Sigma X)}$
Monotone Neigh. Frames	Monotone Bisimilarity	$\mathcal{N}(X)$
$\vdots$	$\vdots$	$\vdots$

Automaton types compose: $F \circ G, F + G, F \times G, \ldots$

<table>
<tr><th>DFA</th><th>Transition system</th><th>Markov chain</th></tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
</tr>
<tr><td>$F(X) = \{\mathsf{F}, \mathsf{T}\} \times X \times X$</td><td>$F(X) = \mathcal{P}_{\mathsf{f}}(X)$</td><td>$F(X) = \{\mathsf{F}, \mathsf{T}\} \times \mathcal{D}(X)$</td></tr>
<tr>
<td>$\mathbf{1} \mapsto (\mathsf{F}, \mathbf{2}, \mathbf{3})$
$\mathbf{2} \mapsto (\mathsf{F}, \mathbf{4}, \mathbf{3})$
$\mathbf{3} \mapsto (\mathsf{F}, \mathbf{5}, \mathbf{3})$
$\mathbf{4} \mapsto (\mathsf{T}, \mathbf{5}, \mathbf{4})$
$\mathbf{5} \mapsto (\mathsf{T}, \mathbf{4}, \mathbf{4})$</td>
<td>$\mathbf{1} \mapsto \{\mathbf{2}, \mathbf{3}, \mathbf{4}\}$
$\mathbf{2} \mapsto \{\mathbf{1}, \mathbf{4}\}$
$\mathbf{3} \mapsto \{\mathbf{3}, \mathbf{4}, \mathbf{5}\}$
$\mathbf{4} \mapsto \{\mathbf{4}, \mathbf{5}\}$
$\mathbf{5} \mapsto \{\,\}$</td>
<td>$\mathbf{1} \mapsto (\mathsf{F}, \{\mathbf{2}\colon \frac{1}{3}, \mathbf{3}\colon \frac{2}{3}\})$
$\mathbf{2} \mapsto (\mathsf{F}, \{\mathbf{2}\colon \frac{1}{2}, \mathbf{4}\colon \frac{1}{2}\})$
$\mathbf{3} \mapsto (\mathsf{F}, \{\mathbf{2}\colon \frac{1}{4}, \mathbf{4}\colon \frac{1}{2}, \mathbf{5}\colon \frac{1}{4}\})$
$\mathbf{4} \mapsto (\mathsf{T}, \{\mathbf{4}\colon 1\})$
$\mathbf{5} \mapsto (\mathsf{F}, \{\mathbf{3}\colon \frac{1}{2}, \mathbf{4}\colon \frac{1}{2}\})$</td>
</tr>
</table>

DFA	Transition system	Markov chain
		
$F(X) = \{\mathsf{F}, \mathsf{T}\} \times X \times X$	$F(X) = \mathcal{P}_{\mathsf{f}}(X)$	$F(X) = \{\mathsf{F}, \mathsf{T}\} \times \mathcal{D}(X)$
$1 \mapsto (\mathsf{F}, 2, 3)$ $2 \mapsto (\mathsf{F}, 4, 3)$ $3 \mapsto (\mathsf{F}, 5, 3)$ $4 \mapsto (\mathsf{T}, 5, 4)$ $5 \mapsto (\mathsf{T}, 4, 4)$	$1 \mapsto \{2, 3, 4\}$ $2 \mapsto \{1, 4\}$ $3 \mapsto \{3, 4, 5\}$ $4 \mapsto \{4, 5\}$ $5 \mapsto \{\,\}$	$1 \mapsto (\mathsf{F}, \{2\colon \frac{1}{3}, 3\colon \frac{2}{3}\})$ $2 \mapsto (\mathsf{F}, \{2\colon \frac{1}{2}, 4\colon \frac{1}{2}\})$ $3 \mapsto (\mathsf{F}, \{2\colon \frac{1}{4}, 4\colon \frac{1}{2}, 5\colon \frac{1}{4}\})$ $4 \mapsto (\mathsf{T}, \{4\colon 1\})$ $5 \mapsto (\mathsf{F}, \{3\colon \frac{1}{2}, 4\colon \frac{1}{2}\})$
$2 \equiv 3, 4 \equiv 5$	$1 \equiv 2, 3 \equiv 4$	$2 \equiv 3 \equiv 5$

What is coalgebraic bisimilarity minimization?

The input:

- a functor $F(X)$ – describes automaton type
- a coalgebra $t : C \to F(C)$ – the automaton

What is coalgebraic bisimilarity minimization?

The input:

- a functor $F(X)$ – describes automaton type
- a coalgebra $t : C \to F(C)$ – the automaton

The output:

- a partition $p : C \to C'$
  – the equivalence classes of bisimilar states
- s.t. $p(x) = p(y) \implies Fp(t(x)) = Fp(t(y))$
- $|C'|$ as small as possible

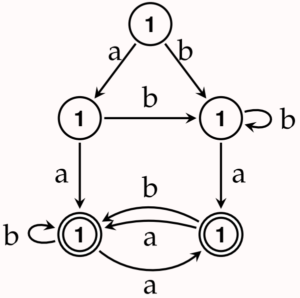
Sketch of our algorithm

1. Assume all states are equivalent
2. Split equivalence classes by *signature* (*normalised* outgoing transitions)
3. Iterate until convergence

Key points

- Only recompute signatures of *changed* states
- Never loop over *unchanged* states

Our algorithm: Minimizing a DFA



Algorithm

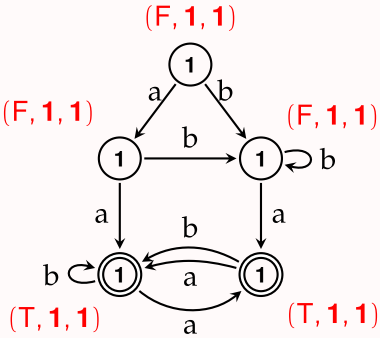
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a DFA



Algorithm

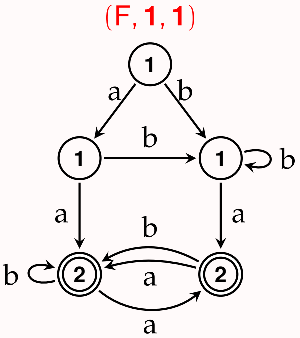
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a DFA



Algorithm

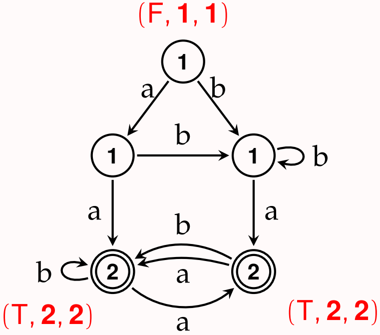
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a DFA



Algorithm

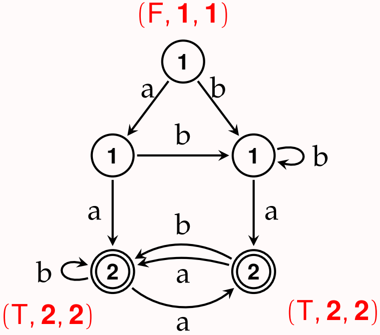
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a DFA



Algorithm

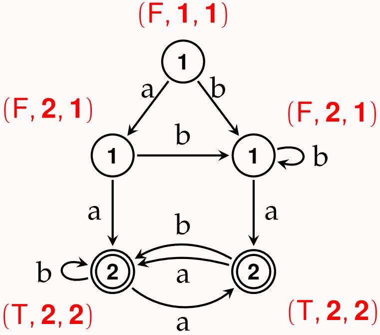
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a DFA



Algorithm

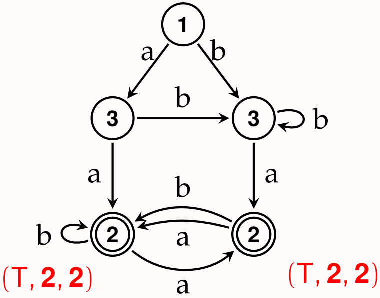
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a DFA



Algorithm

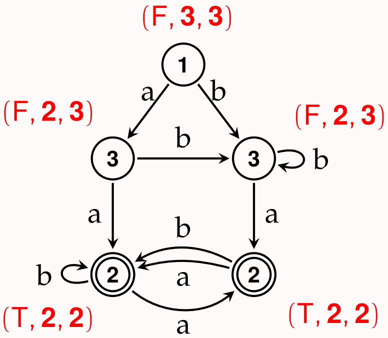
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a DFA



Algorithm

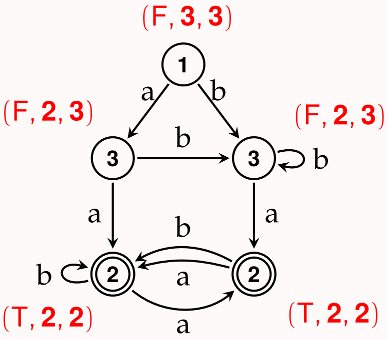
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a DFA



Algorithm

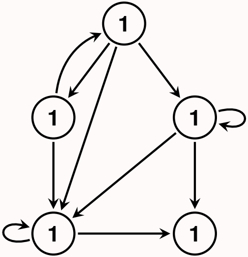
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a transition system



Algorithm

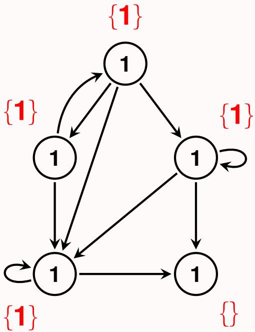
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a transition system



Algorithm

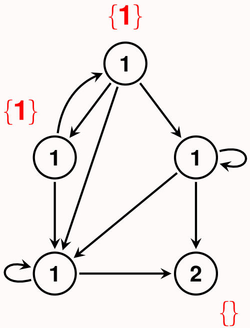
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a transition system



Algorithm

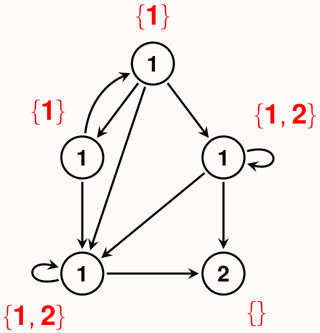
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a transition system



Algorithm

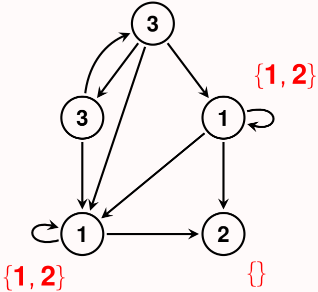
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a transition system



Algorithm

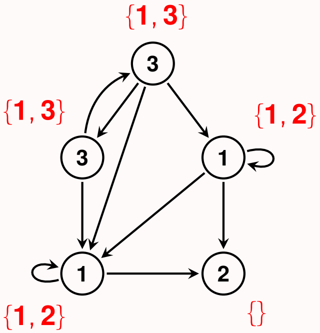
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a transition system



Algorithm

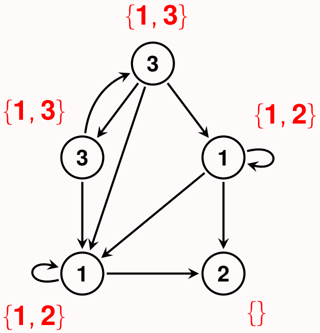
Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

Our algorithm: Minimizing a transition system



Algorithm

Set all the state numbers to 1.

Iterate:

1. Pick equivalence class & compute missing signatures.
2. Assign new state numbers & remove invalid signatures.

Until all states have signatures.

The general picture

1. *Pick equivalence class with missing signatures:*

A	A						
1	1	1	1	1	1	1	1

2. *Compute missing signatures:*

A	A	B	B	B	B	C	A
1	1	1	1	1	1	1	1

3. *Assign new state numbers:*

A	A	B	B	B	B	C	A
2	2	1	1	1	1	3	2

4. *Remove invalid signatures from predecessors*

What we need from the automaton

- Set of states C
- Predecessors of each state $\mathsf{pred} : C \to \mathcal{P}(C)$
- **Procedure to (re)compute signatures**
  $\mathsf{sig} : (C \to \mathbb{N}) \to (C \to F(\mathbb{N}))$

What we need from the automaton

- Set of states C
- Predecessors of each state $\mathsf{pred} : C \to \mathcal{P}(C)$
- **Procedure to (re)compute signatures**
  $\mathsf{sig} : (C \to \mathbb{N}) \to (C \to F(\mathbb{N}))$

Complexity: $O(n^2)$ signature computations

What we need from the automaton

- Set of states C
- Predecessors of each state $\mathsf{pred} : C \to \mathcal{P}(C)$
- **Procedure to (re)compute signatures**
  $\mathsf{sig} : (C \to \mathbb{N}) \to (C \to F(\mathbb{N}))$

Complexity: $O(n^2)$ signature computations

Key: *re-use old state number for*
largest new equivalence class
Invalidates fewer signatures

Complexity: $O(m \log n)$ signature computations

Hopcroft's trick

State	1	2	3	4	5	6	7	8	9
Iteration 1	1	1	1	1	1	1	1	1	1

Hopcroft's trick

State	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Iteration 1	1	1	1	1	1	1	1	1	1
Iteration 2	2	3	3	3	1	1	1	1	1

Hopcroft's trick

State	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Iteration 1	1	1	1	1	1	1	1	1	1
Iteration 2	2	3	3	3	1	1	1	1	1
Iteration 3	2	3	3	3	1	1	4	4	5

Hopcroft's trick

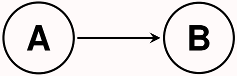
State	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Iteration 1	1	1	1	1	1	1	1	1	1
Iteration 2	2	3	3	3	1	1	1	1	1
Iteration 3	2	3	3	3	1	1	4	4	5
Iteration 4	2	3	3	6	1	1	4	4	5

Hopcroft's trick

State	1	2	3	4	5	6	7	8	9
Iteration 1	1	1	1	1	1	1	1	1	1
Iteration 2	2	3	3	3	1	1	1	1	1
Iteration 3	2	3	3	3	1	1	4	4	5
Iteration 4	2	3	3	6	1	1	4	4	5
Iteration 5	2	3	3	6	1	7	4	4	5

Each state's number changes $O(\log n)$ *times!*

Why $O(m \log n)$ signature recomputations?



An edge **A** $\rightarrow$ **B** may cause a signature recomputation of **A** when **B**'s state number changes.

Time complexity: $O(\Phi_F \cdot m \log n)$

Key ingredient: *never touch unchanged states*

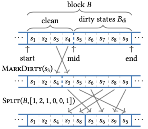
Time complexity: $O(\Phi_F \cdot m \log n)$

Key ingredient: *never touch unchanged states*

- n-way partition refinement data structure
- also tracks invalid signatures
- uses radix sort & bucket sort for n-way split

Algorithm 5 Refinable partition data structure with n-way split

```
procedure MarkDirty(s)
    ▹ Determine the block data ◃
    B := block_of[s]
    j := state2loc[s]
    (start, mid, end) := blocks[B]
    ▹ Do nothing if already dirty ◃
    if mid ≤ j then return
    ▹ Add to worklist if first dirty state ◃
    if mid = end then worklist.add(B)
    ▹ Swap s with the last clean state ◃
    s' := loc2state[mid − 1]
    state2loc[s'] := j
    state2loc[s] := mid
    loc2state[j] := s'
    loc2state[mid] := s
    ▹ Move marker to make s dirty ◃
    blocks[B].mid −= 1
```




```
procedure Split(B, A: B_di → ℕ)
    ▹ Cumulative counts of sub-block sizes ◃
    (start, mid, end) := blocks[B]
    D[0.. max_i A[i] + 1] := 0
    D[0] := mid − start
    for j ∈ B_di do
        D[A[j]] += 1
    i_max = argmax_i D[i]
    for i ∈ 1..|D| do
        D[i] += D[i − 1]
    ▹ Re-order the states by A-value ◃
    dirty := copy(loc2state[mid..end])
    for i ∈ reverse(0..|A|) do
        D[A[i]] −= 1
        j := start + D[A[i]]
        loc2state[j] := dirty[i]
        state2loc[loc2state[i]] := j
    D[0] −= mid − start
    ▹ Create blocks and assign IDs ◃
    D.add(end − start)
    old_block_count := |blocks|
    for i ∈ 0..|D| − 1 do
        j0 := start + D[i]
        j1 := start + D[i + 1]
        if i = i_max then
            blocks[B] = (j0, j1, j1)
        else
            blocks.add(j0, j1, j1)
            idx := |blocks| − 1
            block_of[loc2state[j0..j1]] := idx
    return old_block_count..|blocks|
```

Algorithm 6 Optimized Partition Refinement for all Set functors

```
procedure PartRefSetFun(C, sig, pred)          ▹ i.e. for the implementation of c: C → FC
    Create a new refinable partition structure p: C → ℕ
    Init p to have one block of all states, and all states marked dirty.
    while there is a block B with a dirty state do
        Compute the arrays                                  } Compute signatures,
            (sigs_di: B_di → 2*) := (x ↦ sig(x, p))         } in total O(m log n) calls
            (sigs_cl: B_cl1 → 2*) := (x ↦ sig(x, p))        } to the coalgebra
        A: B_di → ℕ := Renumber'(sigs_di, sigs_cl)          } Split B according to
        B⃗_new := Split(B, A)                                 } signatures in O(|B_di|)
        for every B' ∈ B⃗_new and s ∈ B' do                   } Mark dirty all states with
            for every s' ∈ pred(s) do                       } a successor in a new block
                MarkDirty(s')                               } in total time O(m log n)
    return the partition p
procedure Renumber'(p: B_di → 2*, q : B_cl1 → 2*)
    (A: B_di → ℕ) := Renumber(p)
    if there are d ∈ B_di, c ∈ B_cl1 with p(d) = q(c) then  } Ensure that A(d) = 0 for all dirty
        Swap the values 0 and A(d) in the array A.          } states d that have the same
    return A                                                } signature as the clean states.
```

$\implies$ **signature recomputations dominate**

Comparison

	CoPaR	*DCPR*	*Boa*
Complexity	$O(m \log n)$	$O(\phi_F \cdot n^2)$	$O(\phi_F \cdot m \log n)$
Generality	Zippable	Coalg	Coalg
Language	Haskell	Haskell	Rust

benchmark		time (s)			memory (GB)		
type	**n/10^3**	***CoPaR***	***DCPR***	***Boa***	***CoPaR***	***DCPR***	***Boa***
fms	**1639**	232	84	1.12	16	$0.51_{\times 32}$	0.19
	4459	–	406	4.47	-	$1.69_{\times 32}$	0.58
wlan	**607**	105	855	0.28	16	$0.15_{\times 32}$	0.04
	1632	–	2960	0.79	-	$3.79_{\times 32}$	0.09
wta$_W$	**152**	566	79	0.74	16	$0.64_{\times 32}$	0.08
	944	–	675	11.96	-	$6.79_{\times 32}$	1.23
wta$_Z$	**156**	438	82	0.48	16	$0.68_{\times 32}$	0.09
	1008	–	645	16.75	-	$5.64_{\times 32}$	1.33
wta$_2$	**155**	449	160	0.81	16	$0.62_{\times 32}$	0.08
	1300	–	1377	23.35	-	$7.09_{\times 32}$	1.65

Comparison

	mCRL2	*Boa*
Complexity	$O(m \log n)$	$O(\phi_F \cdot m \log n)$
Generality	LTS+	Coalg
Language	C++	Rust

What is the cost of generality?

What is the cost of generality?

benchmark		time (s)		memory (GB)	
type	**n/10³**	***mCRL2***	***Boa***	***mCRL2***	***Boa***
	2417	13.9	1.4	1.78	0.25
cwi	**7839**	214.2	15.8	5.78	0.81
	33950	282.2	31.5	16.62	2.78
	6021	33.8	3.1	2.12	0.52
vasy	**11027**	51.6	6.1	2.77	0.62
	12324	56.9	7.0	3.10	0.73

For ***mCRL2***, we pick its best algorithm and self-reported time.
For ***Boa***, we report wall-clock time.

We are $O(\phi_F \cdot m \log n)$ rather than $O(m \log n)$, so

Why is it fast?

We are $O(\phi_F \cdot m \log n)$ rather than $O(m \log n)$, so

Why is it fast?

- Read 1 byte from memory: ≈ 200 cycles

We are $O(\phi_F \cdot m \log n)$ rather than $O(m \log n)$, so

Why is it fast?

- Read 1 byte from memory: ≈ 200 cycles
- Read next 63 bytes from memory: ≈ 1 cycle
- Computation (arithmetic, bit ops): ≈ 1 cycle

We are $O(\phi_F \cdot m \log n)$ rather than $O(m \log n)$, so

Why is it fast?

- Read 1 byte from memory: ≈ 200 cycles
- Read next 63 bytes from memory: ≈ 1 cycle
- Computation (arithmetic, bit ops): ≈ 1 cycle

ϕ_F *is cheap!*

Don't incrementalize, just recompute:

- Saves memory
- Saves random reads
- Saves iterations*

Conclusion

Minimization can be **simple**, **generic**, and **fast**

Conclusion

Minimization can be **simple**, **generic**, and **fast**

Future

- Other notions of equivalence (*e.g.*, branching)
- Specialization by monomorphisation
- Integration into Storm (with Sebastian Junges)
- Minimize *your* automata!