### O(m log n) Time Algorithms for DFA Minimization and More

**Antti Valmari**  
Tampere University of Technology

<table>
<thead>
<tr>
<th>Part I: DFA Minimization</th>
<th>Part II: and More</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Deterministic Finite Automata</td>
<td>10 Bisimilarity</td>
</tr>
<tr>
<td>2 Minimization of Deterministic</td>
<td>11 Minimal Bisimilar Graph</td>
</tr>
<tr>
<td>3 Block Splitting</td>
<td>12 Paige-Tarjan $O(m \log n)$ RCP Alg.</td>
</tr>
<tr>
<td>5 Gries’ Data Structures (Roughly)</td>
<td>14 Markov Chains and Markov Dec.</td>
</tr>
<tr>
<td>6 More Recent … Data Structure</td>
<td></td>
</tr>
<tr>
<td>7–9 DFA Min… in $O(m \log n)$ Time</td>
<td>15 Conclusions</td>
</tr>
</tbody>
</table>

---

**AV**  
$O(m \log n)$ Time ...  
2010-10-09  
0/15
Part I:

DFA Minimization
1 Deterministic Finite Automata

- \( D = (Q, \Sigma, \delta, \hat{q}, F) \)
  - \( Q \) = states
  - \( \Sigma \) = labels (the alphabet)
  - \( \delta \) = transitions (\textit{partial function}!)
  - \( \hat{q} \) = initial state
  - \( F \) = final states

- Let \( n = |Q|, m = |\delta| = |\text{defined transitions}|, \alpha = |\Sigma| = |\text{available labels}| \) 
  - technical convenience assumption: \( n = O(m) \) \( (\text{e.g., } n \leq 2m + 1) \)

- Let \( q \in Q \)

- The \textbf{language} accepted by \( q \) is the set of strings of labels on the paths from \( q \) to final states
  - e.g., bottom middle state: \( \{aba, aaba, \ldots, ba, baaba, \ldots\} \)
  - e.g., bottom right state: \( \{\varepsilon, aba, aaba, aaaba, \ldots, \ldots\} \)

- Denote it with \( \mathcal{L}(q) \)

- The language accepted by \( D \) is \( \mathcal{L}(D) = \mathcal{L}(\hat{q}) \)
2 Minimization of Deterministic Finite Automata

- **The minimization problem:**
  Find the smallest DFA that accepts the same language as the given DFA.

- **solution:**
  1. Remove *irrelevant* states and transitions. (textbook stuff)
     - those that are not reachable from $\hat{q}$
     - those from which no final state can be reached (except $\hat{q}$)
  2. Merge states that accept the same language.

- How can we test if $\mathcal{L}(q_1) = \mathcal{L}(q_2)$?
3 Block Splitting

- States are partitioned into blocks
- $q_1$ and $q_2$ go to different blocks only when it is certain that $\mathcal{L}(q_1) \neq \mathcal{L}(q_2)$
- Initially blocks are $F$ and $Q \setminus F$ (or just one of them, if the other is $\emptyset$)
- Blocks are split as long as possible
- Reason for putting $q_1$ and $q_2$ to different blocks: for some label $a$ and block $B$, $q_1$ has and $q_2$ does not have an $a$-transition to a state in $B$
- At most $n - 1$ successful splittings ($n = |\text{States}|$)
- Problem: vulnerable to lots of work

- $O(n^2)$ even if $\alpha = 1$ and $m = O(n)$
- Cost of “useless” “little” work may be important
4 Hopcroft’s Ideas [1971]

- Clarified and improved by Gries [1973] (and, e.g., Knuutila [2001])
- Assumes that $\delta$ is full
- **Idea:** Traverse transitions backwards
  - splitter = (block, label) = $(B, a)$
  - process one splitter at a time
  - find the $q$ such that $\delta(q, a) \in B$, move them to tentative new blocks
  - each block splits to backwards-encountered and others (if both non-empty)
  $\Rightarrow$ no futile scanning of states without relevant output transitions

- **Idea:** If $(B, a)$ has been used and $B$ splits to $B_1$ and $B_2$, then it suffices to use one of $(B_1, a)$ and $(B_2, a)$
  - use the “smaller” one (meaning of “smaller” is less trivial than it seems!)
  $\Rightarrow$ each state is used as a splitter state at most $\log_2 n$ instead of $n$ times
- Both ideas $\Rightarrow$ running time in $O(\alpha n \log n)$ $m \leq \alpha n$, often $m \ll \alpha n$
5 Gries’ Data Structures (Roughly)

- Partition of $Q$
  - for each block, there is a doubly linked list of the states in it
  - the block has a pointer to its main list and “tentative new” list
  - each state has a pointer to its block
  - the block knows its size (and the size of the “tentative new” list)

- Inverse transitions: for each $q$ and $a$, the states $q'$ such that $\delta(q', a) = q$

- Worksets for temporary storage
  - unprocessed splitters, and pointers to them for each $(B, a)$
  - backwards-encountered states — don’t cut the branch on which you sit!
  - backwards-encountered blocks = touched blocks

- $\Theta(\alpha n)$ memory
  while there are unprocessed splitters $(B, a)$ do
  choose any and remove it from the workset
  compute its backwards-encountered states
  use backwards-encountered states to tentatively split blocks
  for each touched block $B'$ do
  split or reset back to earlier status
  if $B'$ is split then for $b \in \Sigma$ do update the $(B', b)$
6 More Recent Refinable Partition Data Structure

- Maintains a partition of \{1, 2, \ldots, N\}, for some \(N\)
- Like Gries, constant time \(\text{Mark}(e)\) and amortized constant time \(\text{Split}(s)\)

\[
\begin{array}{c}
\text{set} & \text{set} & \text{set} & \text{set} \\
\end{array}
\]

\[
\begin{array}{cccc}
elem & \vdots & \text{marked} & \text{unmarked} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{loc}[e] & \text{first}[s] & \text{mid}[s] & \text{end}[s] \\
\end{array}
\]

- Also \(\text{sidx}[e]\)
- Of course, all arrays must be updated appropriately in each operation

- \(\text{Mark}(e)\) swaps \(e\) with the first unmarked element and increments \(\text{mid}[s]\)
  - **Trick 1 for the future:** returns set number iff all elements were unmarked

- **Trick 2 for the future:** \(\text{Split}(s)\) gives new block number to smaller half
  - earlier papers: marked states become the new block
  - does not affect amortized speed, as long as new \(\neq\) unmarked bigger half
7 DFA Minimization in $O(m \log n)$ Time 1/3

- Valmari & Lehtinen [2008], improvements Valmari [2010]
- **Problem**: How to avoid spending excessive time scanning empty splitters?
  - empty $(B, a) = \text{no } a\text{-transition ends at } B$
  - and how to avoid using and initializing $\Theta(\alpha n)$ memory?
  $\Rightarrow$ cannot use, e.g., `in_trans[ state, label ]`
- **Idea**: Non-empty splitters constitute a partition of transitions that can be maintained similarly to blocks
  $\Rightarrow$ Two refinable partitions in the same program
  - $B = \text{blocks, partition of states}$
  - $C = \text{cords, partition of transitions}$
- Inverse transitions: $In\_trans[q] = \{ (q_1, a, q_2) \in \delta \mid q_2 = q \}$
  - numbers of input transitions of $q$ *in arbitrary order*
  $\Rightarrow$ no need for $\Theta(\alpha n)$ data structures, easy to initialize
- Worksets: only one, $W$
  - e.g., array of integers used as a stack
8 DFA Minimization in $O(m \log n)$ Time 2/3

• Algorithm *in great detail*

\[
c := 1; \ b := 2; \ W := \emptyset;
\]

**while** $c \leq |C|$ **do**

**use cord** \#\(c\) **to split blocks**

\[
c := c + 1; \ W := \emptyset
\]

**while** $b \leq |B|$ **do**

**use block** \#\(b\) **to split cords**

\[
b := b + 1; \ W := \emptyset
\]

• Earlier Trick 1

$\Rightarrow$ \(b'\) and \(c'\) may be added to \(W\) without testing if they are already there

• Why is there no workset for unprocessed block-splitters and cord-splitters?

  – Hopcroft, Gries: either smaller or new half must be added to unprocessed
  – Trick 2: new number is given to smaller half $\Rightarrow$ these cases are the same
  – unprocessed are chosen for processing in first in – first out order

$\Rightarrow$ the unprocessed are always \(\{b, b + 1, \ldots, |B|\}\) and \(\{c, c + 1, \ldots, |C|\}\)
Problem: Not enough time to initially sort the transitions! \(O(m \log m)\)
- we aim at so fast an algorithm that ordinary things become too slow
- solution: counting sort + classification trick of Aho & al. [1974] exercise
- practical average-time solution: hash table
- engineer’s: don’t bother, \(O(m \log m)\) is not much worse than \(O(m \log n)\)

A prototype implementation
- 590 lines of C++ (+ libraries for formatted i/o and error messages)
- refinable partition data structure: 50 lines, other data structures: 20 lines
- block splitting: 40 lines
- Aho & al. [1974] & heapsorting as an alternative: 60 lines
- removal of irrelevant states: 70 lines, other initialization: 80 lines
- input and output: 130 lines
- heapsort: 40 lines, range-checking array: 60 lines, main comment: 40 lines

\(\Rightarrow\) Compared to general idea of the difficulty of programming fast DFA minimization, this is very simple

- 50 sec on a laptop when \(n = 10^5\), \(\alpha = 1000\), \(m = 5 \cdot 10^6\) (includes i/o)
Part II:

and More
10 Bisimilarity

- Weakest relation that satisfies:
  - \( q_1 \sim q_2 \Rightarrow \text{label}(q_1) = \text{label}(q_2) \)
  - \( q_1 \sim q_2 \land (q_1, a, q'_1) \in \Delta_1 \Rightarrow \exists q'_2 : q'_1 \sim q'_2 \land (q_2, a, q'_2) \in \Delta_2 \)
  - \( q_1 \sim q_2 \land (q_2, a, q'_2) \in \Delta_2 \Rightarrow \exists q'_1 : q'_1 \sim q'_2 \land (q_1, a, q'_1) \in \Delta_1 \)
  - \( q_1^{\text{init}} \sim q_2^{\text{init}} \), or \( \forall q_1 \in \text{Initials}_1 : \exists q_2 \in \text{Initials}_2 : q_1 \sim q_2 \), and ...

- Fundamental relation in concurrency theory

- Often much better “strong” equivalence than isomorphism
  - unifies states that only differ on \( x \), if the next thing on \( x \) is \( x := 0 \)
11 Minimal Bisimilar Graph

- Nondeterministic version of DFA minimization
- Remove what is not reachable from initial states
- Fuse bisimilar states
- Applications
  - smaller graph for further processing
  - bisimilarity test
- **Problem** due to nondeterminism: three-way splitting

![Diagram of bisimilar graph minimization process]
12 Paige–Tarjan $O(m \log n)$ RCP Algorithm [1987]

- **Relational coarsest partition problem:**
  Nondeterministic graph, only one label ($\alpha = 1$)

- **Idea:** Compound blocks, “$q$-$B$-counters”, and “$q$-counters”

- Compound block $\approx$
  union of blocks that has been used for splitting
  - let $\hat{B}$ be the compound block that covers block $B$

- Biggest block in a compound block
  need not be used in further splitting
  $\Rightarrow$ each state is used in a splitter at most $\log_2 n$ times

- **Problem:** How to implement three-way splitting?

  $\Rightarrow$ Maintain, for each $q$ and $B$, the number of transitions from $q$ to $B$

- Each $B'$ has (at most) three kinds of sub-blocks
  - **left block:** $\#(q, B) > 0 = \#(q, \hat{B} \setminus B)$
  - **middle block:** $\#(q, B) > 0 < \#(q, \hat{B} \setminus B)$
  - **right block:** $\#(q, B) = 0$
13 Extension to Bisimulation [2009, 2010]

- (Still RCP) cannot afford to represent counters that store 0
  ⇒ data structure trickery!

- Extend $q$-$B$-counters to $q$-$a$-$B$-counters
- Use cords (and the year 2010 DFA tricks)
- Initialization: sort each cord according to start states
  - $O(m_a \log m_a)$, where $m_a \leq n^2$
- Splitting
  - distinguish left blocks from middle blocks by counter values
  - extract left blocks like with DFAs
  - update old and create new counter when extracting middle block
  - main loop and splitting of cords are like with DFAs
- Tricky details here and there and in the correctness proof, but it works!
These ideas have also been applied to state lumping of Markov Chains by Valmari and Franceschinis [2010]—would be a story of its own—finds use for the elegant but mostly useless majority candidate algorithm.

Everything has been put together in a program for minimizing Markov Decision Processes.
Part III:

Conclusions
15 Conclusions

• $O(m \log n)$ time DFA, bisimulation, and MDP minimization are possible
  – such an algorithm for DFAs was found amazingly late, 1971 ↔ 2008
  – the other two problems are strictly more general
  – $O(m \log n)$ time Markov chain lumping was solved in 2003, but our results simplify it

• The breakthrough was the use of another partition, this time of transitions

• In the end the programs are relatively simple
  – we got rid of some data structures in earlier algorithms

• However, many tricky ideas had to be fine-tuned to make it all work
  – algorithms, correctness proofs, and programs
  – details are important for obtaining the promised performance!

• Hidden theme: representing a mapping where some result value is far more common than others
  – sparse mapping
  – avoid explicitly representing that value
Part IV:

Thank you for attention.

Questions?