Directly constructing minimal DFAs : combining two algorithms by Brzozowski

Directly Constructing Minimal DFAs: Combining Two Algorithms by Brzozowski

Bruce W. Watson

Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa, watson@cs.up.ac.za

Abstract

In this paper, we combine (and refine) two of Brzozowski's algorithms — yielding a single algorithm which constructs a minimal deterministic finite automaton (DFA) from a regular expression.

Keywords: *Brzozowski, deterministic finite automata, minimization, automaton construction, algorithmics*
Computing Review Categories: *D.1.4, E.1, F.2.2, G.2.2*

1 Introduction

To obtain a minimal DFA, an implementor usually codes separate construction and minimization algorithms, composing them (at runtime) to yield the minimal DFA. A single, combined algorithm, which both constructs the DFA and simultaneously minimizes it is likely to be even more efficient thanks to the fact that fewer intermediate data structures are built. Recent examples of this phenomenon can be found in [4, 3, 8, 9, 11, 15, 16] which all present algorithms that construct automata while maintaining minimality or near minimality. Those algorithms have been shown to out-perform the naïve runtime composition of a construction algorithm with a minimization algorithm.

In this paper, the two algorithm design focii are:

1. The real life performance of the algorithm. The asymptotic running time is usually a complex function of the inherent complexity of the input regular expression, which we do not discuss further here.

2. The quality of the output automaton — in this case, the result is a *minimal* automaton.

This paper is structured as follows:

- §2 presents the two component algorithms.
- §3 combines the two algorithms into one.
- §4 gives an extended example.
- §5 presents benchmarking data on the resulting algorithm's performance.
- §6 presents the paper's closing remarks.

Preliminary results of this work were reported at the 2000 Conference on Implementations and Applications of Automata [12].

1.1 Related work

There are numerous possible minimizing DFA construction algorithms, at least one for each possible combination of a construction algorithm with a minimization algorithm. (Using [10] as a basis, this yields at least two hundred possibilities.) The algorithm presented here is a relatively elegant combination of two algorithms which themselves are simple and easily presented. The resulting algorithm is easy to manipulate and refine, and therefore easy to optimize and implement. Preliminary benchmarking indicates that the new algorithm out-performs a runtime composition of the two component algorithms.

1.2 Preliminaries

We assume that the reader is familiar with elementary finite automata theory, definitions and principles, including: states, transitions, determinism, minimality, regular expressions, language denoted by a regular expression, derivatives (of a regular expression), and *similarity* of regular expressions. (In the remainder of this paper, all derivatives are actually a shorthand for their similarity equivalence classes.) Readers who are not familiar with this material should consult any one of the standard textbooks, for example [7, 17].

One definition which is not always presented in textbooks is that of *reversal*. We can define function[1] reverse which reverses:

- a string, returning a string with the letters of the original string in the reverse order;
- a language, by returning the language consisting of the reversal of each string in the original language;
- a regular expression; this can be defined inductively on the structure of regular expressions; and

[1] Despite being called a function, it would likely be implemented as an imperative program.

- a finite automaton, by returning a new automaton in which the direction (source and destination) of each transition is reversed, start states (in the original automaton) are made into final states, and final states (in the original automaton) are made into start states.

We define subset to be the *subset construction* — a function which takes a finite automaton and returns a DFA with no unreachable states, accepting the same language (see, for example, [7]). We do not elaborate further on the subset construction.

All of the algorithms presented here are in the guarded command language (with some additional annotations for comments), a type of pseudo code — see [5].

2 The component algorithms

We begin with the construction portion of such an algorithm. From [10, Chapter 6], there are at least twenty known construction algorithms; unfortunately, the performance data presented in [10, Chapter 14] does not include all of those algorithms. Nonetheless, anecdotal experience (also from industrial applications such as in computational linguistics) indicates that a good choice is the *derivatives*-based algorithm by Brzozowski [2] (we refer to this algorithm as Brzconstr). That algorithm, which yields a DFA, is easy to present and to implement, satisfying the first of our two goals. Furthermore, it can be made more efficient[2] by providing a regular expression *simplifier*, since each state (in the resulting automaton) is represented by a regular expression. Such a simplifier is then able to identify two states which may otherwise have been distinguished, for example, the states $a \cdot b$ and $a \cdot \varepsilon \cdot b$ would be identified if the simplifier is aware of the identity $E \cdot \varepsilon \equiv E$. (In order for Brzozowski's construction to terminate (correctly), the simplifier must at least recognize similarity; we do not discuss this further in this paper.) Brzozowski's construction algorithm (taking a regular expression E) is given[3] in Algorithm 2.1 on page 19.

We now focus on the selection of a minimization algorithm. There are several known minimization algorithms — see [10, Chapter 7]. Presently, Hopcroft's algorithm is the algorithm with the best-known asymptotic running time (of $O(n \log n)$, for n the number of states). Unfortunately, that algorithm is also one of the more difficult algorithms to present, manipulate, refine, and implement — cf. that David Gries's paper [6] shed significant light on the algorithm's derivation. As we discuss in [14, 13], Brzozowski's minimization algorithm [1] has also proven to be fast in practice, frequently out-performing Hopcroft's algorithm. (For a performance comparison, see [10, Chapter 15].) Without presenting the details (which can be found in [10, Chapter 7]), Brzozowski's minimization algorithm is defined as follows (for automaton M, where we may choose to implement the 'functions' as imperative programs): $\mathsf{Brzmin}(M) = \mathsf{subset}(\mathsf{reverse}(\mathsf{subset}(\mathsf{reverse}(M))))$.

[2] Extensive benchmarking data for this is not available, although, industry applications of the algorithm have shown that it is more efficient than other popular algorithms such as the Aho-Sethi-Ullman algorithm [10].

[3] Q refers to the set of states; $S, F \subseteq Q$ are the sets (respectively) of start and final states; δ is the transition relation (mapping a state and an alphabet symbol to a state); and $\mathcal{L}$ is an overloaded function giving the language of a regular expression or finite automaton.

3 Combining algorithms

In this section, we combine the two algorithms by Brzozowski to yield a more efficient single algorithm than their runtime composition. Initially, we consider expression $\mathsf{Brzmin}(\mathsf{Brzconstr}(E))$. Expanding the definition of Brzmin, we get $\mathsf{subset}(\mathsf{reverse}(\mathsf{subset}(\mathsf{reverse}(\mathsf{Brzconstr}(E)))))$. Straightforward refinements and improvements of this algorithm can be obtained by combining algorithm components which are adjacent in terms of composition, and for the moment we focus on the inner components. We can make an important observation: reverse commutes with all construction algorithms mapping a regular expression to a finite automaton. This allows us to switch the two innermost components of the composition: $\mathsf{subset}(\mathsf{reverse}(\mathsf{subset}(\mathsf{Brzconstr}(\mathsf{reverse}(E)))))$. The outermost two components of Brzmin (specifically, $\mathsf{subset}(\mathsf{reverse}(\cdots))$) only requires that its input is a DFA accepting the language denoted by $\mathsf{reverse}(E)$. Since Brzconstr already yields a DFA, the invocation of subset following Brzconstr is redundant, and we simplify it to $\mathsf{subset}(\mathsf{reverse}(\mathsf{Brzconstr}(\mathsf{reverse}(E))))$. To make further improvements, we switch to the imperative version of the algorithm shown in Algorithm 3.1 on page 20.

In that algorithm, we begin by reversing the regular expression E. Thanks to the symmetry of derivatives and regular expressions, we have an alternative: use *right derivatives* instead of left derivatives. This yields Algorithm 3.2 on page 20 (in which the changes from the previous algorithm have been underlined).

Our remaining goal is to combine the final reversal (of the DFA) in the last statement with the preceding portion of the algorithm. Finite automaton reversal is as simple as exchanging the places of the start and final states (S and F), and reversing the update of the transition function (δ), yielding Algorithm 3.3 on page 21. This is our final algorithm. We will not manipulate the remaining composition further (since it does not yield any readability advantages), nor will we explicitly present subset.

4 An example

In this section, we discuss an example of a finite automaton built with the new algorithm. We focus on the intermediate automata as well as the end-result and contrast this with the

Algorithm 2.1 (Brzconstr):

---

func Brzconstr(E) $\rightarrow$
$Q,\delta,S,F := \emptyset,\emptyset,\{E\},$
$\emptyset;$
$done, to\_do := \emptyset, S;$
do $to\_do \neq \emptyset \rightarrow$
  let p be some state such that $p \in to\_do$;
  $done, to\_do := done \cup \{p\}, to\_do \setminus \{p\}$;
  $destination := a^{-1}p$ — the left derivative of p by a;
  if $destination \notin done \rightarrow$
    { *destination*'s out-transitions are still to be built }
    $to\_do := to\_do \cup \{destination\}$
  [] $destination \in done \rightarrow$ skip
  fi;
  $Q := Q \cup \{destination\}$;
  { make a transition from p to *destination* on a }
  $\delta(p,a) := destination$;
  if $\varepsilon \in \mathcal{L}(destination) \rightarrow$
    { *destination* should be a final state }
    $F := F \cup \{destination\}$
  [] $\varepsilon \notin \mathcal{L}(destination) \rightarrow$ skip
  fi
od;
{ $\mathcal{L}(Q,\delta,S,F) = \mathcal{L}(E)$ }
return (Q,δ,S,F)
cnuf

□

Algorithm 3.1:

$Q,\delta,S,F := \emptyset,\emptyset,\{\text{reverse}(E)\}$,
$\emptyset$;
done, *to_do* $:= \emptyset, S$;
do *to_do* $\neq \emptyset \rightarrow$
  let p be some state such that $p \in$ *to_do*;
  done, *to_do* $:=$ *done* $\cup \{p\}$, *to_do* $\setminus \{p\}$;
  destination $:= a^{-1}p$ — the left derivative of p by a;
  if *destination* $\notin$ *done* $\rightarrow$
    { *destination*'s out-transitions are still to be built }
    to_do $:=$ *to_do* $\cup \{$*destination*$\}$
  $[]$ *destination* $\in$ *done* $\rightarrow$ skip
  fi;
  $Q := Q \cup \{$*destination*$\}$;
  { make a transition from p to *destination* on a }
  $\delta(p,a) :=$ *destination*;
  if $\varepsilon \in \mathcal{L}($*destination*$)$ $\rightarrow$
    { *destination* should be a final state }
    $F := F \cup \{$*destination*$\}$
  $[]$ $\varepsilon \notin \mathcal{L}($*destination*$)$ $\rightarrow$ skip
  fi
od;
$\{\ \mathcal{L}(Q,\delta,S,F) = \mathcal{L}(\text{reverse}(E))\ \}$
$(Q,\delta,S,F) := \text{subset}(\text{reverse}(Q,\delta,S,F))$;
$\{\ \mathcal{L}(Q,\delta,S,F) = \mathcal{L}(E)\ \}$

□

Algorithm 3.2:

$Q,\delta,S,F := \emptyset,\emptyset,\{E\}$,
$\emptyset$;
done, *to_do* $:= \emptyset, S$;
do *to_do* $\neq \emptyset \rightarrow$
  let p be some state such that $p \in$ *to_do*;
  done, *to_do* $:=$ *done* $\cup \{p\}$, *to_do* $\setminus \{p\}$;
  destination $:= pa^{-1}$ — the right derivative of p by a;
  if *destination* $\notin$ *done* $\rightarrow$
    { *destination*'s out-transitions are still to be built }
    to_do $:=$ *to_do* $\cup \{$*destination*$\}$
  $[]$ *destination* $\in$ *done* $\rightarrow$ skip
  fi;
  $Q := Q \cup \{$*destination*$\}$;
  { make a transition from p to *destination* on a }
  $\delta(p,a) :=$ *destination*;
  if $\varepsilon \in \mathcal{L}($*destination*$)$ $\rightarrow$
    { *destination* should be a final state }
    $F := F \cup \{$*destination*$\}$
  $[]$ $\varepsilon \notin \mathcal{L}($*destination*$)$ $\rightarrow$ skip
  fi
od;
$\{\ \mathcal{L}(Q,\delta,S,F) = \mathcal{L}(\text{reverse}(E))\ \}$
$(Q,\delta,S,F) := \text{subset}(\text{reverse}(Q,\delta,S,F))$;
$\{\ \mathcal{L}(Q,\delta,S,F) = \mathcal{L}(E)\ \}$

□

Algorithm 3.3:

$Q, \delta, \underline{F}, \underline{S} := \emptyset, \emptyset,$
$\{E\}, \emptyset;$
$done, to\_do := \emptyset, \underline{F};$
do $to\_do \neq \emptyset \rightarrow$
  let p be some state such that $p \in to\_do$;
  $done, to\_do := done \cup \{p\}, to\_do \setminus \{p\}$;
  $destination := pa^{-1}$ — the right derivative of p by a;
  if $destination \notin done \rightarrow$
    { *destination*'s out-transitions are still to be built }
    $to\_do := to\_do \cup \{destination\}$
  [] $destination \in done \rightarrow$ skip
  fi;
  $Q := Q \cup \{destination\}$;
  { make a transition from *destination* to p on a }
  $\delta(destination, a) := p$;
  if $\varepsilon \in \mathcal{L}(destination) \rightarrow$
    { *destination* should be a start state }
    $\underline{S} := \underline{S} \cup \{destination\}$
  [] $\varepsilon \notin \mathcal{L}(destination) \rightarrow$ skip
  fi
od;
$\{ \mathcal{L}(Q, \delta, S, F) = \mathcal{L}(E) \}$
$(Q, \delta, S, F) := \text{subset}(Q, \delta, S, F);$
$\{ \mathcal{L}(Q, \delta, S, F) = \mathcal{L}(E) \}$

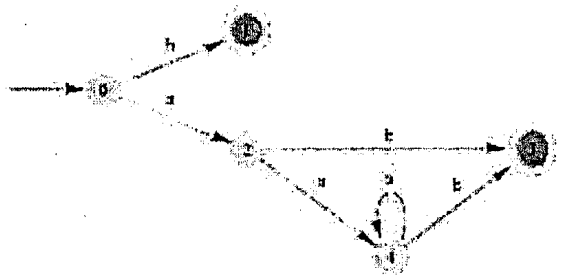
□

intermediate automata in the runtime composition of Brzozowski's algorithms.

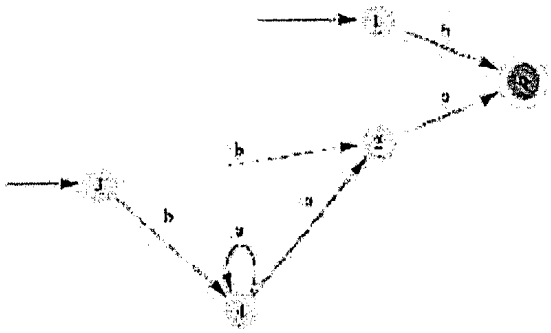
The regular expression considered is $E = a^*b$. Below, we have the states obtained under Brzconstr (the union symbol, $\cup$, being used to represent alternation in the regular expressions) and left-derivatives for E (where derivatives leading to the sink-state are not introduced, since they are useless states):

- State 0: regular expression = a^*b; not a final state; a derivative = $\varepsilon a^*b \cup \emptyset$; b derivative = $\emptyset a^*b \cup \varepsilon$.
- State 1: regular expression = $\emptyset a^*b \cup \varepsilon$; final state; no a or b derivatives.
- State 2: regular expression = $\varepsilon a^*b \cup \emptyset$; not a final state; a derivative = $\emptyset a^*b \cup \varepsilon a^*b \cup \emptyset$; b derivative = $\emptyset a^*b \cup \varepsilon \cup \emptyset$.
- State 3: regular expression = $\emptyset a^*b \cup \varepsilon \cup \emptyset$; final state; no a or b derivatives.
- State 4: regular expression = $\emptyset a^*b \cup \varepsilon a^*b \cup \emptyset$; not a final state; a derivative = $\emptyset a^*b \cup \varepsilon a^*b \cup \emptyset$; b derivative = $\emptyset a^*b \cup \varepsilon \cup \emptyset$.

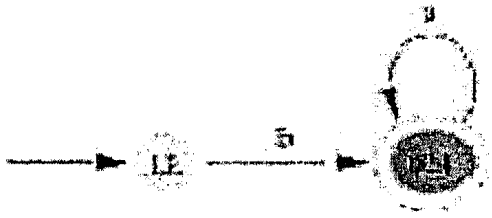
The resulting automaton is:



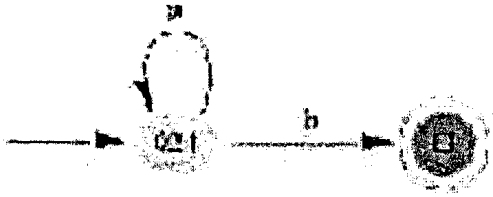
Following reverse we have:



Following subset we get:



Again following reverse we get:



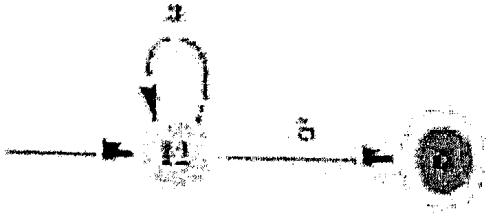
This remains unchanged following the final application of subset.

We can now turn to the intermediate result of our new algorithm. We obtain the following states and right-derivatives for E (again, not building right derivatives which give a sink-state):

- State 0: regular expression = a^*b; not a start state; no a derivative; b derivative = $a^*\varepsilon$.
- State 1: regular expression = $a^*\varepsilon$; start state; a derivative = $a^*\varepsilon \cup a^*\emptyset$; no b derivative.
- State 2: regular expression = $a^*\varepsilon \cup a^*\emptyset$; start state; a derivative = $a^*\varepsilon \cup a^*\emptyset$; no b derivative.

The resulting automaton is:

The final application of subset gives:



5 Algorithm performance

Small-scale benchmarking of this algorithm has already been performed, yielding the following two observations:

1. The algorithm is only a *constant factor* improvement over the sequential composition of the uncombined component algorithms.
2. On average, the algorithm performs 26% faster than the sequential composition of the component algorithms. At this time, insufficient data is available to determine the variance or standard deviation in general.

6 Closing comments

The derived algorithm has the following characteristics:

- The derivation of the algorithm is easily understood and the correctness is easily established.
- Thanks to the easily-understood derivation, the algorithm is also quickly implemented.
- The final algorithm is as easy to implement as the original Brzconstr. The differences can be factored, leaving a common algorithmic skeleton. (This technique is used in [10] for keyword pattern matching algorithms with a common skeleton.)
- One of the algorithmic components, the subset construction, is usually already implemented in automata toolkits.
- The component algorithms have excellent performance in practice. It follows that the new algorithm will display similar (if not better) performance.

Acknowledgements: I would like to thank Nanette Y. Saes and the anonymous referee for improving the quality of this paper.

References

[1] Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. volume 12 of *MRI Symposia Series*, pages 529–561, Polytechnic Institute of Brooklyn, 1962. Polytechnic Press.

[2] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[3] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April 2000.

[4] Jan Daciuk, Bruce W. Watson, and Richard E. Watson. Incremental construction of minimal acyclic finite state automata and transducers. In Lauri Karttunen and Kemal Oflazer, editors, *Proceedings of the International Workshop on Finite State Methods in*

Natural Language Processing, pages 48–56, Ankara, Turkey, June 1998.

[5] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[6] David Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[7] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[8] Stoyan Mihov. *Direct Building of Minimal Automaton for Given List*. PhD thesis, Bulgarian Academy of Science, 1999.

[9] Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92:181–189, 1992.

[10] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, September 1995.

[11] Bruce W. Watson. A fast new semi-incremental algorithm for the construction of minimal acyclic DFAs. In Derick Wood and Denis Maurel, editors, *Proceedings of the Third Workshop on Implementing Automata*, volume 1660 of *Lecture Notes in Computer Science*, pages 91–98, Rouen, France, September 1998. Springer-Verlag.

[12] Bruce W. Watson. Combining two algorithms by Brzozowski. In Wood and Yu [18], pages 242–249.

[13] Bruce W. Watson. A history of Brzozowski's DFA minimization algorithm. In Wood and Yu [18].

[14] Bruce W. Watson. A history of Brzozowski's DFA minimization algorithm. Technical report, Department of Computer Science, University of Pretoria, South Africa, 2001.

[15] Bruce W. Watson. A new recursive algorithm for building minimal acyclic deterministic finite automata. Technical report, Department of Computer Science, University of Pretoria, South Africa, 2001.

[16] Bruce W. Watson. A new recursive algorithm for building minimal acyclic deterministic finite automata. In Carlos Martin-Vide and Victor Mitrana, editors, *Grammars and Automata for String Processing: From Mathematics and Computer Science to Biology and Back*. 2002.

[17] Derick Wood. *Theory of Computation*. Harper & Row, 1987.

[18] Derick Wood and Sheng Yu, editors. *Proceedings of the Fifth Conference on Implementations and Applications of Automata*, London, Canada, July 2000.