16

Combining Regular Expressions with Near-Optimal Automata in the FIRE Station Environment

BRUCE W. WATSON, MICHIEL FRISHERT AND LOEK CLEOPHAS

We discuss a method for efficiently computing deterministic Brzozowski (derivatives) automata. Our approach is based on efficiently storing regular expressions using parse trees and expressions using common subexpression elimination.

16.1 Introduction

Derivatives of regular expressions were first introduced by Brzozowski in (Brzozowski, 1964). By recursively computing all derivatives of a regular expression, a deterministic automaton can be constructed. To guarantee convergence of this process, derivatives are compared modulo *similarity*, i.e. modulo associativity, commutativity, and idempotence of the union operator. Additionaly, through simplification based on the identities for regular expressions, the number of derivatives can be further reduced.

We have developed an efficient method for computing such automata by combining parse trees with the automata. In our implementation, we recognize and remove similar regular expressions through *global common subexpression elimination* (GCSE) on the parse tree. The concept of GCSE is a well-known optimization technique in the field of compilers, see for example (Cocke, 1970). Because the regular expressions are stored in parse trees,

Inquiries into Words, Constraints and Contexts.
Antti Arppe et al. (Eds.)

and subtrees of common expressions are reused optimally, we can avoid the expense of storing an entire regular expression (as a string or parse tree) per derivative. Also, we never compute derivatives twice for a class of similar regular expressions.

Reduction of the number of regular expressions by identities is done through regular expression rewriting. Due to the generic framework for rewriting, we are able to reduce using additional rewrite rules, which results in smaller automata.

An earlier version of some of the research reported on in this paper was presented as a poster paper at CIAA 2004 (Frishert and Watson, 2004).

16.1.1 Historical note by Bruce Watson

These algorithms, data-structures, and techniques have now been implemented in the FIRE Station environment, also described in a number of recent articles. The FIRE Station, related to the FIRE Engine series of toolkits for finite automata and regular expressions, is a workstation-type environment (in software) manipulating regular expressions, finite automata, and other finite-state objects, including their languages. In the mid-1990's, I made several visits to Kimmo in Helsinki. The need and underlying ideas for FIRE Station grew directly out of those brainstorming sessions with Kimmo and his group. There were already a number of tools (notably tools from Xerox and AT&T and also INTEX) available, though all rather tightly coupled to their applications in NLP. The core philosophy behind the FIRE Station is to provide a number of efficient application-neutral algorithms and data-structures. Layered on top of this core will be the option of several 'skins,' providing the look-and-feel of the various application domains for finite state techniques, such as: NLP, modeling of concurrent systems, compiler design, text indexing and hardware design. Each such skin may additionally provide domain-specific operators, views of the automata, etc. Kimmo's ongoing interest and inputs have given a unique NLP perspective on the potential applications of a tool such as FIRE Station. (FIRE Station is being made available—also in source form—for noncommercial use.)

16.2 Preliminaries

Definition 1 [Regular Languages and Regular Expressions] We define regular expressions RE over alphabet Σ and the languages they denote, $\mathcal{L}_{RE} \in RE \rightarrow \mathcal{P}(\Sigma^*)$ as follows:

- $\emptyset \in RE$ and $\mathcal{L}_{RE}(\emptyset) = \emptyset$
- $\varepsilon \in RE$ and $\mathcal{L}_{RE}(\varepsilon) = \{\varepsilon\}$
- For all $a \in \Sigma, a \in RE$ and $\mathcal{L}_{RE}(a) = \{a\}$

For $E, F \in RE$

- $E|F \in RE$ and $\mathcal{L}_{RE}(E|F) = \mathcal{L}_{RE}(E) \cup \mathcal{L}_{RE}(F)$
- $E \cdot F \in RE$ and $\mathcal{L}_{RE}(E \cdot F) = \mathcal{L}_{RE}(E) \cdot \mathcal{L}_{RE}(F)$
- $E^* \in RE$ and $\mathcal{L}_{RE}(E^*) = \mathcal{L}_{RE}(E)^*$ □

16.3 Parse Trees

The parse tree is a tree based representation of regular expressions. Each node in the tree defines a regular expression based on its children and the operator associated with the node. In contrast with the binary parse trees that are often found in the literature, our parse trees are n-ary trees. Nodes in the parse tree are represented by the set V, and each node $v \in V$ is either an internal node (has children), or a leaf node.

Definition 2 [The Set of Regular Operators] We define the set of constants and operations on regular languages by their names:
$operators = \{[\emptyset], [\varepsilon], [\Sigma], [*], [|], [\cdot]\}$ □

Definition 3 [Regular Operator Nodes] The set of nodes V is partitioned over $operators$:

- $(\forall i : i \in operators : V_i \subseteq V)$
- $(\cup i : i \in operators : V_i) = V$
- $(\cap i : i \in operators : V_i) = \emptyset$ □

Definition 4 [Structure of the Parse Tree] The structure of the parse tree is uniquely determined by the following four functions:

- $symbol : V_{[\Sigma]} \rightarrow \Sigma$.
- $term : V_{[*]} \rightarrow V$
- $termset : V_{[|]} \rightarrow \mathcal{P}(V)$
- $termlist : V_{[\cdot]} \rightarrow V^*$ □

Definition 5 [Parse Tree to Regular Expression] For a node $v \in V$, we define a mapping $regex \in V \rightarrow RE$, from parse tree to regular expression straightforwardly as:

- $v \in V_{[\emptyset]} \Rightarrow regex(v) = \emptyset$
- $v \in V_{[\varepsilon]} \Rightarrow regex(v) = \varepsilon$
- $v \in V_{[\Sigma]} \Rightarrow regex(v) = symbol(v)$
- $v \in V_{[*]} \Rightarrow regex(v) = term(v)^*$
- $v \in V_{[|]} \Rightarrow regex(v) = (|w : w \in termset(v) : w)$
- $v \in V_{[\cdot]} \Rightarrow regex(v) = termlist(v)_0 \cdot \ldots \cdot termlist(v)_{|termlist(v)|-1}$ □

Definition 6 [Regular Language of a Node $\mathcal{L}_{PT}(v)$] For a node $v \in V$, the regular language represented by v is given by $\mathcal{L}_{PT}(v) \in V \rightarrow RE$ as follows:
$\mathcal{L}_{PT}(v) = \mathcal{L}_{RE}(regex(v))$ □

Definition 7 [Creating Regular Expression Nodes] We can create new nodes in the parse tree through the function $mknode \in ([\emptyset] \cup [\varepsilon] \cup ([\Sigma] \times \Sigma) \cup ([*] \times V) \cup ([|] \times \mathcal{P}(V)) \cup ([\cdot] \times V^*)) \rightarrow V$. This function has to satisfy the following specification:

- $\mathcal{L}_{PT}(mknode([\emptyset])) = \emptyset$
- $\mathcal{L}_{PT}(mknode([\varepsilon]))) = \{\varepsilon\}$
- $\mathcal{L}_{PT}(mknode([\Sigma], a)) = \{a\}, (\forall a \in \Sigma)$
- $\mathcal{L}_{PT}(mknode([*], v)) = \mathcal{L}_{PT}(v)^*, (\forall v \in V)$
- $\mathcal{L}_{PT}(mknode([|], W)) = (\bigcup w : w \in W : \mathcal{L}_{PT}(w)), (\forall W \in \mathcal{P}(V))$
- $\mathcal{L}_{PT}(mknode([\cdot], W)) = \mathcal{L}_{PT}(W_0) \cdot \ldots \cdot \mathcal{L}_{PT}(W_{|W|-1}), (\forall W \in V^*)$ □

Note that these specifications seem weaker than they need to be, however, they allow room for refinement in the implementation. For example, given nodes $v, w \in V$, so that $\mathcal{L}_{PT}(v) = \{\varepsilon\}$ and $\mathcal{L}_{PT}(w) = \{a\}$, the function $mknode([|], v, w)$ may return a new node $u \in V_{[|]} \wedge termset(u) = \{v, w\}$; but it may also simply return w. This leaves room for improvements that will be discussed at a later point.

16.4 Derivatives

First, we adapt Brzozowski's definition of derivatives to our parse tree.

Definition 8 Function $\delta \in V \rightarrow RE$ determines whether or not the regular language represented by a node $v \in V$ contains the empty string ε and is defined as:

$\delta(v) = \varepsilon$, if $\varepsilon \in \mathcal{L}_{PT}(v)$
$\delta(v) = \emptyset$, if $\varepsilon \notin \mathcal{L}_{PT}(v)$ □

Definition 9 [Brzozowski Derivatives] For node $v \in V$ and symbol $a \in \Sigma$ the derivatives function $D \in V \times \Sigma \rightarrow RE$ is defined as:

- if $v \in V_{[\emptyset]}$, then $D(v, a) = \emptyset$
- if $v \in V_{[\varepsilon]}$, then $D(v, a) = \emptyset$
- if $v \in V_{[\Sigma]} \wedge symbol(v) = a$, then $D(v, a) = \varepsilon$
- if $v \in V_{[\Sigma]} \wedge symbol(v) \neq a$, then $D(v, a) = \emptyset$
- if $v \in V_{[*]}$, then $D(v, a) = D(term(v), a) \cdot v$
- if $v \in V_{[|]}$, then $D(v, a) = (|u : u \in childset(v) : D(u, a))$
- if $v \in V_{[\cdot]}$, then $D(v, a) =$
  $(D(termlist(v)_0, a) \cdot termlist(v)_1 \cdot \ldots \cdot termlist(v)_{|termlist(v)|-1})$
  $|(\delta(termlist(v)_0) \cdot D(termlist(v)_1 \ldots termlist(v)_{|termlist(v)|-1}, a))$ □

Our goal is to find or create a node in the parse tree that represents the derivative of a given node-symbol pair. To this end, we introduce the function $\Delta \in V \times \Sigma \rightarrow V$. A straightforward Δ could satisfy $regex(\Delta(v, a)) =$

$D(v,a)$. We deviate slightly and only require the weaker condition of $\mathcal{L}_{PT}(\Delta(v,a)) = \mathcal{L}_{RE}(D(v,a))$, i.e. that the regular languages rather than the regular expressions are equivalent. This allows us some room to add optimizations, potentially leading to smaller automata. It also allows for a straightforward definition of Δ in terms of $mknode$, as seen in Def. 10.

Definition 10 [Derivatives via the Parse Tree] We create the function $\Delta \in V \times \Sigma \to V$, which computes the node representing the derivative of a given node $v \in V$ and symbol $a \in \Sigma$, such that $\mathcal{L}_{PT}(\Delta(v,a)) = \mathcal{L}_{RE}(D(v,a))$. Δ is expressed in terms of $mknode$:

- if $v \in V_{[\emptyset]}$, then $\Delta(v,a) \equiv mknode([\emptyset])$
- if $v \in V_{[\varepsilon]}$, then $\Delta(v,a) \equiv mknode([\emptyset])$
- if $v \in V_{[\Sigma]} \wedge a = symbol(v)$, then $\Delta(v,a) \equiv mknode([\varepsilon])$
- if $v \in V_{[\Sigma]} \wedge a \neq symbol(v)$, then $\Delta(v,a) \equiv mknode([\emptyset])$
- if $v \in V_{[*]}$, then $\Delta(v,a) \equiv mknode([\cdot], \Delta(term(v),a), v)$
- if $v \in V_{[|]}$, then $\Delta(v,a) \equiv mknode([|], \cup u \in V : u \in termset(v) : \Delta(u,a))$
- if $v \in V_{[\cdot]} \wedge termlist(v)_0 \notin null$, then $\Delta(v,a) \equiv mknode([\cdot], \Delta(termlist(v)_0,a), termlist(v)_1, \ldots, termlist(v)_{|termlist(v)|-1})$
- if $v \in V_{[\cdot]} \wedge termlist(v)_0 \in null$, then $\Delta(v,a) \equiv mknode([|], \{mknode([\cdot], \Delta(termlist(v)_0,a), termlist(v)_1, \ldots, termlist(v)_{|termlist(v)|-1}), \Delta(mknode([\cdot], termlist(v)_1, \ldots, termlist(v)_{|termlist(v)|-1}), a)\})$

□

All that remains is an implementation of the function $mknode$. To this end, we now discuss our means of dealing with similar expressions and reduction via identities.

16.5 Common Subexpression Elimination

The subexpression of a node v is the regular expression as described by the parse tree. It is not uncommon for two equivalent subexpressions to occur in different parts of the parse tree. By finding and eliminating these *common subexpressions*, we can merge similar derivatives.

Definition 11 [Subexpression Equivalence $\sim_{cse}$] Nodes $v, w \in V$ are in relation $v \sim_{cse} w$ holds if any of the following holds:

- $v = w$
- $v, w \in V_{[\emptyset]}$
- $v, w \in V_{[\varepsilon]}$
- $v, w \in V_{[\Sigma]} \wedge symbol(v) = symbol(w)$
- $v, w \in V_{[*]} \wedge term(v) \sim_{cse} term(w)$

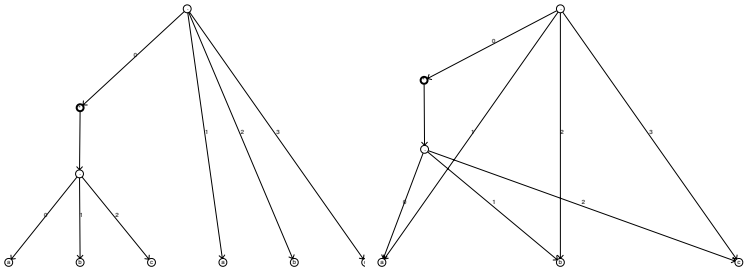

FIGURE 1 (a) $(abc)^*abc$ before GCSE (b) after GCSE

- $v, w \in V_{[|]} \wedge (\forall p \in termset(v) : (\exists q \in termset(w) : p \sim_{cse} q)) \wedge$ $(\forall q \in termset(w) : (\exists p \in termset(v) : p \sim_{cse} q))$
- $v, w \in V_{[\cdot]} \wedge (\forall i : 0 \leq i \leq |termlist(v)| : termlist(v)_i \sim_{cse} termlist(w)_i)$

Note that $v \sim_{cse} w \equiv regex(v) = regex(w)$ □

We can reduce all nodes that are in the same equivalence class defined by $\sim_{cse}$ to a single node. This process is called *Global Common Subexpression Elimination* (GCSE). Removing equivalent nodes does not affect the regular languages represented, however, it does change the parse tree into a directed acyclic graph (DAG). As an example of this, the regular expression *(abc)*abc* results in the parse tree in Figure 1(a). The subexpression *abc* occurs in two locations. We can replace these by a single instance, as in Figure 1(b). Note that we will continue to use the term parse tree, since that is still the intended interpretation of the graph; the fact that it is a DAG merely provides us with a more efficient representation.

If we integrate GCSE into the function $mknode$, we can establish the following invariant:

Definition 12 [CSE Invariant] $(\forall v, w \in V : v \sim_{cse} w \Rightarrow v = w)$ □

This CSE invariant means that we will never create a new node if a $\sim_{cse}$ equivalent node already exists, and it allows us to detect common subexpressions without resorting to expensive recursion:

Definition 13 [Subexpression Equivalence without recursion] Nodes $v, w \in V$, are in relation $v \sim_{cse} w$ if CSE Invariant of Def. 12 holds, and if any of the following holds:

- $v = w$
- $v, w \in V_{[\emptyset]}$
- $v, w \in V_{[\varepsilon]}$
- $v, w \in V_{[\Sigma]} \wedge symbol(v) = symbol(w)$

TABLE 1 Rewrite Rules for identities. Note that $E \in RE$

$$\emptyset \cdot E \rightarrow \emptyset$$
$$E \cdot \emptyset \rightarrow \emptyset$$
$$\varepsilon \cdot E \rightarrow E$$
$$E|\emptyset \rightarrow E$$

- $v, w \in V_{[*]} \wedge term(v) = term(w)$
- $v, w \in V_{[|]} \wedge termset(v) = termset(w)$
- $v, w \in V_{[.]} \wedge termlist(v) = termlist(w)$ □

When attempting to create a new (internal) node with operator $o \in operators$ and children W, finding a node that is $\sim_{cse}$ equivalent (if it exists) can be done by examining the parents for the child nodes in W to find a node $m \in V_o$ and children equal to W. We can instantly find the parents of a particular node by storing the reverse relations from the parse tree. Note that it is sufficient to search the parents of only one of the elements of W for an equivalent parent node, rather than all the children, because a matching node will be parent to all the nodes in W. To maintain the CSE Invariant of Def. 12, we return the equivalent node if it is found to exist, and only create a new node if it does not exist.

16.6 Rewriting

As suggested by Brzozowski (Brzozowski, 1964), the number of derivatives can be reduced by simplification using the identities. We implement this using a rewriting system as described in (Frishert et al., 2003). As discussed, the specification of Δ was deliberately weak, which now allows us to use any number of rewrite rules. If we wish to obtain the exact Brzozowski derivatives automata, we restrict ourselves to the rewrite rules in Table 1. If we add additional rewrite rules we can potentially obtain smaller automata.

Combining rewriting and GCSE, the function $mknode$ can now be implemented as follows for a given operator and operand (either a symbol, node, nodeset or a nodelist): If an applicable rewrite rule exists, apply that rule, resulting in a new operator/operand pair. Repeat this until there are no further applicable rewrite rules. For the final operator/operand pair, we search the existing nodes for a CSE-equivalent node. If such a node exists, we return that node; otherwise we add a new node to V and set its operator/operands accordingly.

16.7 Results and Future Work

We have implemented the approach discussed in this paper in our tool FIRE STATION, see (Frishert, 2005). All figures in this paper were generated using

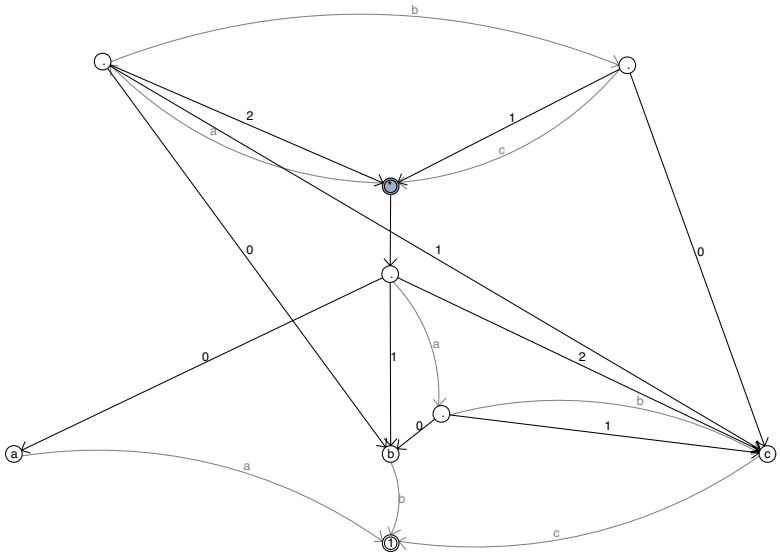

FIGURE 2 Combined Parse Trees for the Derivatives of $(abc)^*$

FIRE STATION. In Figure 2, the combined parse graph for the derivatives of $(abc)^*$ is shown. The numbered edges indicate the order of concatenated nodes: due to GCSE, a node can be used in multiple concatentations, and the order for these concatentations is sometimes conflicting, making it impossible for the concatenated nodes to be drawn in left-to-right order.

The extended regular operators: negation, intersection, relative/symmetric difference, negation, as well as the POSIX character classes, and repeat ranges can easily be added to this framework and require no special treatement.

The approach we have discussed in this paper also lends itself well to partial derivatives (Antimirov, 1996), which also have been implemented successfully in FIRE STATION.

We see two interesting next steps. First, additional rewrite rules, which may result in further reduction of automata sizes, could be included. Second, it may be possible to perform incremental minimization, reducing intermediate memory requirements.

References

Antimirov, V. 1996. Partial derivatives of regular expressions and finite automata

constructions. *Theoretical Computer Science* 155:291–319.

Brzozowski, J.A. 1964. Derivatives of Regular Expressions. *Journal of the ACM* 11(4):481–494.

Cocke, J. 1970. Global common subexpression elimination. In *Proceedings of a symposium on compiler optimization*, pages 20–24.

Frishert, Michiel. 2005. *FIRE Station: a FInite automata & Regular Expression playground*. Master's thesis, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven.

Frishert, M., L. Cleophas, and B.W. Watson. 2003. The effect of rewriting regular expressions on their accepting automata. In *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA 2003)*, vol. 2759 of *Lecture Notes in Computer Science*, pages 304–305. Springer.

Frishert, M. and B.W. Watson. 2004. Combining regular expressions with (near-) optimal Brzozowski automata. In *Proceedings of the 9th International Conference on Implementation and Application of Automata (CIAA 2004)*, vol. 3317 of *Lecture Notes in Computer Science*, pages 319–320. Springer.