On Regular Expression Hashing to Reduce FA Size

Wikus Coetser, Derrick G. Kourie, and Bruce W. Watson

Fastar Research Group, Department of Computer Science, University of Pretoria
spring.haas.meester@gmail.com, dkourie@cs.up.ac.za, bruce@bruce-watson.com

Abstract. In [7], a new version of Brzozowski's algorithm was put forward which relies on regular expression hashing to possibly decrease the number of states in the generated finite state automata. This method utilizes a hash function to decide which states are merged, but does not, in general, construct *-equivalence classes on automaton states, as is done in minimization algorithms. The consequences of this approach depends on the hash function used, and include the construction of a super-automaton and potential non-determinism. A revised version of the hashing algorithm in [7] is presented that constructs a deterministic automaton. A method for rewriting the hash function input is presented that allows the construction of a hash function that is an injection, mapping a unique integer to each regular language. A method for measuring the difference between the exact- and super-automaton is presented.

Keywords: finite state automaton, DFA, NFA, state merging, equivalence classes, regular languages, super-automaton, approximate automaton, hash function, minimization, exact automaton, sub-automaton

1 Introduction

Brzozowski has a well-known algorithm for deriving a finite automaton from a regular expression. In [7], a modified version of Brzozowski's Algorithm was presented for constructing an *approximate automaton*, hereafter referred to as a *super–automaton*. By merging the states in the event of hash function clashes, the resulting super-automaton may have fewer states than the finite state automaton[1] that would have been generated by the original algorithm.

The results of this merging process are explored in this article: in section 2, the original and modified versions of Brzozowski's Algorithm are presented. In section 3, a proof is given that the approach in section 2 always produces a super–automaton. In section 4, it is shown why the approach in section 2 may lead to non–determinism, and a new algorithm is put forward for constructing a deterministic automaton. In section 5.1 a method is put forward for judging the relative quality of super- and exact automata. In section 5.3, a method is given for modifying the input of the hash function in order to allow the entire language of a state to be taken into account when hashing. In section 5.4 the effect of the modulo function used in most hash functions is considered.

2 Brzozowski's Algorithm with state merging

In order to understand how state merging is implemented with a hash function, it is necessary to first look at Brzozowski's original algorithm, given in Algorithm 1, and taken from [7].

[1] Finite state automaton is abbreviated FA, a non-deterministic FA is abbreviated NFA and a deterministic FA is abbreviated DFA.

Wikus Coetser, Derrick G. Kourie, Bruce W. Watson: On Regular Expression Hashing to Reduce FA Size, pp. 227–241.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1

Brzozowski's Algorithm takes a regular expression, and constructs a finite state automaton from that expression, using left derivatives[1], first symbol sets and a test for whether a regular language given by a regular expression contains the empty string:[2]

- The left derivative of a regular expression RE with respect to an input symbol s is written $s^{-1}RE$, and represents all the strings of the regular language defined by RE, with their respective first symbols removed.
- The first symbol set of a regular expression RE, written as $first(RE)$, is the set containing the first symbol of each string represented by the regular expression RE.
- Given that ε represents the empty string and that $L(RE)$ represents the set of strings of the language described by the regular expression RE, the test $\varepsilon \in L(RE)$ is written $nullable(RE)$.

Note that in Algorithms 1, 2 and 3, an FA is represented by a 4-tuple $\langle Q, \delta, E, F \rangle$, where Q is the set of states of the automaton, δ is the state transition function, E is the initial state and F is the set of final states. The alphabet Σ is not referenced in any of the respective algorithms and should therefore be regarded as implicit in the FA's representation.

The method used by Brzozowski's Algorithm is illustrated in Figure 1: regular expressions are associated with states in the algorithm.

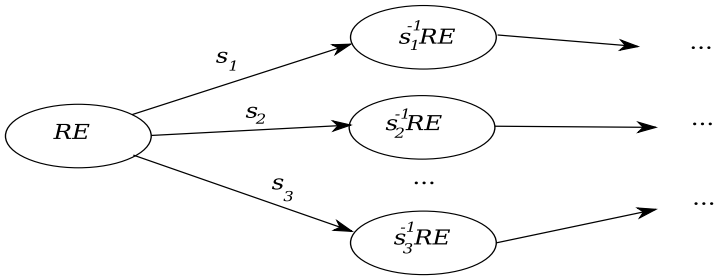

Figure 1. Brzozowski's Algorithm without state merging

Given the input regular expression RE, the first symbol set of RE is calculated. In Figure 1 the first symbol set $\{s_1, s_2, s_3\}$ corresponds to the out-transition symbols of the state marked as RE. For each first symbol s, the left derivative $s^{-1}RE$ is calculated. This left derivative represents the next state for s. The remap function in Algorithm 1 (Brzozowski's original algorithm) maps each regular expression to the next unassigned integer. If a regular expression re-appears (taking idempotence, associativity and commutativity of regular expression operators into account) as a result of computing the derivatives, a cycle forms in the automaton, representing a plus or star closure. When a regular expression represents a regular language that contains an empty string, a final state has been reached.

In the remainder of this text, $state(RE)$ is used to designate a state associated in some unspecified (i.e. abstract) way with the regular expression RE. If we wish to emphasise that in some concrete implementation, the association of the state with the

[2] The notation for first symbols, left derivatives and nullable regular expressions is taken from [3].

Algorithm 1 (Brzozowski's Algorithm with Remapping)

```
func Brz(RE_init)
    next, δ, F, remap := 0, ∅, ∅, ∅;
    remap[RE_init], next := next, next + 1;
    done, todo := ∅, {RE_init};
    do todo ≠ ∅ →
        let RE_j be some regular expression such that RE_j ∈ todo;
        done, todo := done ∪ {RE_j}, todo\{RE_j};
        { Only expand out-transitions for symbols in the first symbol set ofRE_j }
        for s : first(RE_j) →
            { Use the left derivatives to calculate the next state }
            destination := s^-1 RE_j
            if destination ∉ done ∪ todo →
               { Update the todo set in order to expand the automaton for destination }
               todo := todo ∪ {destination};
               remap[destination], next := next, next + 1
            [] destination ∈ done ∪ todo → skip
            fi
            ; δ(remap[RE_j], s) := remap[destination]
        rof
        ;
        if nullable(RE_j) →
           { The final states all have a right language containing the empty string }
           F := F ∪ {remap[RE_j]}
        [] ¬nullable(RE_j) → skip
        fi
    od;
    return ⟨{0, ..., next − 1}, δ, 0, F⟩
cnuf
```

regular expression is via a function, for example $hash$, then the notation $hash(RE)$ is used.

Consider two regular expressions RE_1 and RE_2. Jointly, these two regular expressions might form $state(RE_1)$ and $state(RE_2)$ in some FA, denoted by F. Suppose $L(RE_1) = \{ab\}$, $L(RE_2) = \{cd\}$ and $L(F) = \{ab, cd\}$ respectively. If F had been built by Algorithm 1, then states $remap(RE_1)$ and $remap(RE_2)$ respectively would have been constructed, and the languages associated with these states, as well as with F would be preserved.

In the Algorithm 2, hashing is used to assign an integer to each state, instead of the remap function. If the two regular expressions RE_1 and RE_2 hash to the same integer, then the collision is not resolved. Instead, the states are merged. This is illustrated in Figure 2. Note that the language of the automaton with the states merged is $\{ab, ad, cd, cb\}$, which is different from $L(F)$, given above as $\{ab, cd\}$.

This merging behaviour has two consequences: the construction of a super-automaton, and the automaton becoming potentially non-deterministic. These consequences are discussed in the next two sections.

3 Super-automata and exact automata

Previously in [7], approximate automata were described informally as being the output of Algorithm 2. Here the preferred nomenclature of *super-automata* will be used

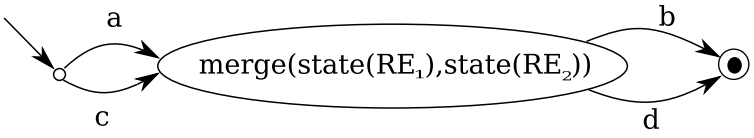

Figure 2. Brzozowski's Algorithm with state merging: $state(RE_1)$ and $state(RE_2)$ have been merged

Algorithm 2 (Brzozowski's Algorithm with Hashing — NFA version)

func $Brz\_hash\_NFA(RE_{init})$
  $Q, \delta, F := \emptyset, \emptyset, \emptyset$;
  $done, todo := \emptyset, \{RE_{init}\}$;
  do $todo \neq \emptyset \rightarrow$
    let RE_j be some regular expression such that $RE_j \in todo$;
    $done, todo, h := done \cup RE_j, todo \backslash RE_j, hash(RE_j)$;
    $Q := Q \cup \{h\}$;
    { Only expand out-transitions for symbols in the first symbol set ofRE_j }
    for $s : first(RE_j) \rightarrow$
      { Use the left derivatives to calculate the next state }
      $destination := s^{-1}RE_j$;
      if $destination \notin done \cup todo \rightarrow$
        { Update the todo set in order to expand the automaton for $destination$ }
        $todo := todo \cup destination$
      [] $destination \in done \cup todo \rightarrow$ **skip**
      fi
      ; $\delta(h, s) := \delta(h, s) \cup \{hash(destination)\}$;
    rof
    ;
    if $nullable(RE_j) \rightarrow$
      { The final states all have a right language containing the empty string }
      $F := F \cup \{h\}$
    [] $\neg nullable(RE_j) \rightarrow$ **skip**
    fi
  od;
  return $\langle Q, \delta, hash(RE_{init}), F \rangle$
cnuf

instead of approximate automata. The notion of a super-automaton of a regular language is formally defined, and it is then formally shown that the output of Algorithm 2 is a super-automaton of the regular language associated with its input regular expression.

Let RL denote an *intended* regular language, for example the regular language described by the regular expression RE_{init} which is to form the input for Brzozowski's Algorithm. The definition of an exact automaton is:

Definition 1. *If RL is a regular language and FA is an automaton for which $L(FA) = RL$, then FA is an exact automaton of RL.*

The definition for a super-automaton is:

Definition 2. *If RL is a regular language and FA is an automaton for which $L(FA) \supseteq RL$, then FA is a super-automaton of RL.*

For the sake of completeness, the definition of a sub-automaton is also given, even though it does not play a direct role in this article.

Definition 3. *If RL is a regular language and FA is an automaton with $L(FA) \subseteq RL$, then FA is a sub-automaton of RL.*

As can be seen from the definitions above, a super-automaton accepts the same language as an exact automaton, and (possibly) also additional strings. The proof that Algorithm 2 produces a super-automaton of its input regular expression is presented next. Note that $\overrightarrow{L}(s)$ represents the right language of a state s, and $\overleftarrow{L}(s)$ represents the left language.

Theorem 4 (The construction of a super-automaton). *Algorithm 2 produces a super-automaton, FA^s, of $L(RE_{init})$, i.e. $L(FA^s) \supseteq L(RE_{init})$.*
Proof *Consider any two states s_1 and s_2 of $L(RE_{init})$. The language of s_1 is $L(s_1) = \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1)$ and similarly, the language of s_2 is $L(s_2) = \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2)$. If Algorithm 2 merges these states into one called $merge(s_1, s_2)$, then its language is:*

$$L(merge(s_1, s_2)) = (\overleftarrow{L}(s_1) \cup \overleftarrow{L}(s_2)) \cdot (\overrightarrow{L}(s_1) \cup \overrightarrow{L}(s_2))$$

Distributing $\cdot$ over $\cup$ gives

$$\begin{aligned} & \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_2) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2) \\ \supseteq \quad & \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2) \\ = \quad & L(s_1) \cup L(s_2) \end{aligned}$$

Since $L(merge(s_1, s_2)) \supseteq L(s_1) \cup L(s_2)$ for any two states s_1 and s_2 that are merged by Algorithm 2, it follows that $L(FA^s) \supseteq L(RE_{init})$.

Note that the notion of proper set containment does not play a role in theorem 4. Therefore it does not exclude the possibility that Algorithm 2 may produce an automaton FA^s that has the same language as the initial regular expression RE_{init}. What is particularly noteworthy is that this equality may hold even if two or more states are merged. An example of this is when $\overrightarrow{L}(s_1) = \overrightarrow{L}(s_2)$ and $\overleftarrow{L}(s_1) = \overleftarrow{L}(s_2)$, i.e. when the states being merged have the same language. In that case, Algorithm 2 partially fulfills the role of minimizing the output of Algorithm 1.

4 Non-determinism arising from the hashing algorithm

One of the consequences of merging states is that the resulting automaton may, under rather special circumstances, be non-deterministic. To see where non-determinism arises, consider two regular expressions RE_i and RE_j that represent *different* regular languages. Suppose that

$\exists\, s : (first(RE_j) \cap first(RE_i)) \cdot L(s^{-1}RE_i) \neq L(s^{-1}RE_j)$

Suppose also that the hash function used in Algorithm 2 hashed RE_i and RE_j to the same value, but hashed $s^{-1}RE_i$ and $s^{-1}RE_j$ to different values.

Example 5. Let $RE_i = sb, RE_j = sc, s^{-1}RE_i = b$ and $s^{-1}RE_j = c$ and let $hash(RE_i) = hash(RE_j)$ but let $hash(s^{-1}RE_i) \neq hash(s^{-1}RE_j)$. Assume that s is the only first symbol in RE_i and RE_j. In this case Algorithm 2 will construct the automaton shown in Figure 3. Note that, because of the duplicate out-transition for $hash(RE_i)$ (which has been merged with $hash(RE_j)$), the automaton is non-deterministic.

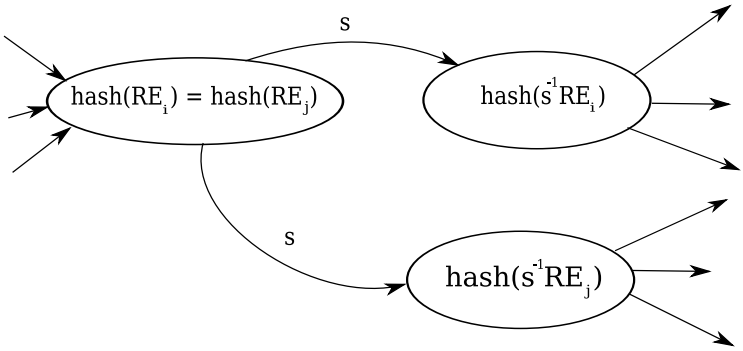

Figure 3. Non-determinism resulting from Algorithm 2

The fact that Algorithm 2 has been designed to generate a non-deterministic automaton is evident from its transition function, δ, that maps from a set of hashed states and transition symbol to a *set* of hashed states.

As is well-known, it is generally preferable to work with a DFA instead of an NFA. The reason for this is that the NFA cannot be represented as a two dimensional state transition table and this has implications for the size of the resulting automaton, as well as for the complexity of the associated FA-related algorithms. Of course, the NFA resulting from Algorithm 2 could be transformed to an equivalent DFA in the normal way. However, in Algorithm 3 a revised version of Algorithm 2 is presented that *directly* constructs a DFA. As in Algorithm 2, the revised algorithm relies on a hash function.

In the remainder of this text we will take the liberty of overloading the union operator, $\cup$. Thus, when used as a regular expression operator, as in $(RE_i \cup RE_j)$, the result is a regular expression such that $L(RE_i \cup RE_j) = L(RE_i) \cup L(RE_j)$. Nevertheless, the semantics of $\cup$ will be clear from the context in which it is used.

Algorithm 3 is premised on the observation that if the non-determinism in the automaton in Figure 3 is to be avoided, then an out-transition on s from a state $hash(RE_j)$ should not be inserted if:

- the state $hash(RE_j)$ corresponds to an existing state $hash(RE_i)$; and
- it is discovered that a transition on s out of state $hash(RE_i)$ already exists.

Indeed, in such an event, the existing transition from state $hash(RE_i)$ (which is equal to $hash(RE_j)$) should be removed. Additionally, a new transition on s from state $hash(RE_i)$ should then be provided to a *new* destination state. The new destination state should now be represented by the hashed value of of the regular expression $(s^{-1}RE_i \cup s^{-1}RE_j)$. The resulting automaton is shown in Figure 4. Note that the resulting automaton is a super-automaton of the regular language of the NDF that would have been constructed by Algorithm 2. It is therefore also a super-automaton of the regular language of the input regular expression to the algorithm.

The pseudocode of Algorithm 3 that appears below relies on the following:

- In the algorithm, the inverse mapping of $hash$ is used, and is called $regex$ Thus if RE is a regular expression such that $hash(RE) = p$, then $regex(p) = RE$. While one cannot in general rely on a hash function having an inverse, if the $hash(RE)$

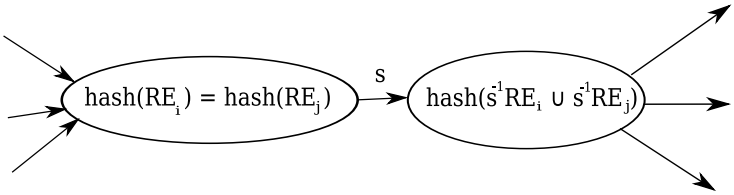

Figure 4. Determinism resulting from Algorithm 3

has previously been computed as p, it is a simple matter to store a backward reference to RE from the stored p.
- In the algorithm, the transition function δ is treated as a set of pairs of the form $\langle\langle h, s\rangle, d\rangle$, i.e. the first element of the pair is itself a pair. In this case, h is the hashed value of a regular expression corresponding to a source state, and d is the hashed value to the regular expression corresponding to the destination state when the symbol s is encountered.
- In order to update δ to account for a newly discovered transition from h to d upon input symbol s, a set union operation is used to augment the set that currently represents δ by an additional element. The form of the operation is thus: $\delta \cup \{\langle\langle h, s\rangle, d\rangle\}$
- In order to remove from δ a mapping that represents the transition from h to d upon input symbol s, the set difference operation is used. The form of the operation is thus: $\delta \setminus \{\langle\langle h, s\rangle, d\rangle\}$

The foregoing discussion about Algorithm 3 referred to the removal of an existing transition, the creation of a new transition and the creation of a new destination state. Examination of the details of the algorithm's pseudocode will indicate where and how these operations take place. However, both the discussion and the algorithm are silent about what should happen to the existing state labelled $hash(s^{-1}RE_i)$ in Figure 3. Should this state as well as its in- and out-transitions be removed? The answer to this question requires special consideration, which is given now, with reference to the contents of Figure 3.

Firstly, note that in dealing with the hash function clash, the algorithm inserts the regular expression $(s^{-1}RE_j \cup s^{-1}RE_i)$ into the *todo* set. (More specifically, the expression in the code $destination \cup regex(d)$ builds the regular expression $(s^{-1}RE_j \cup s^{-1}RE_i)$, which is then assigned to *destination*, and subsequently inserted into *todo* by computing $todo \cup \{destination\}$.) Thus, in some future iteration of the algorithm, it will be selected and all out-transitions that might have been generated previously on state $hash(s^{-1}RE_i)$ will be generated once more on state $hash(s^{-1}RE_i \cup s^{-1}RE_j)$. This is because $first(s^{-1}RE_i) \subseteq first(s^{-1}RE_i \cup s^{-1}RE_j)$. This means that if the only in-transition into state $hash(s^{-1}RE_i)$ was on s, nothing would be lost by discarding state $hash(s^{-1}RE_i)$ from Q, as well as all transitions out of it, as stored in the δ function.

However, if there were *more* transitions into state $hash(s^{-1}RE_i)$ than simply on s, then the algorithm might no longer generate a super-automaton of the input regular expression, if this state and associated transitions were to be discarded. For this reason, the state $hash(s^{-1}RE_i)$ should not be summarily discarded.

On the other hand, if s was indeed the only in-transition to state $hash(s^{-1}RE_i)$, then not removing this state means that it may become an unreachable state in

Algorithm 3 (Brzozowski's Algorithm with Hashing — DFA version)

func $Brz\_hash\_DFA(RE_{init})$
$Q, \delta, F := \emptyset, \emptyset, \emptyset;$
$done, todo, := \emptyset, \{RE_{init}\};$
do $todo \neq \emptyset \rightarrow$
let RE_j be some regular expression such that $RE_j \in todo$;
$done, todo := done \cup \{RE_j\}, todo \backslash \{RE_j\};$
$h := hash(RE_j);$
$Q := Q \cup \{h\};$
{ expand out-transitions for symbols in the first symbol set of RE_j }
for $s : first(RE_j) \rightarrow$
{ compute the left derivative of RE_j with respect tos }
$destination := s^{-1}RE_j;$
if $(\exists d : \langle\langle h, s\rangle, d\rangle \in \delta) \rightarrow$
$\delta := \delta \setminus \{\langle\langle h, s\rangle, d\rangle\};$
$destination := destination \cup regex(d)$
[] $(\nexists d : \langle\langle h, s\rangle, d\rangle \in \delta) \rightarrow$ **skip**
fi
;
if $destination \notin done \cup todo \rightarrow$
$todo := todo \cup \{destination\}$
[] $destination \in done \cup todo \rightarrow$ **skip**
fi
;$\delta := \delta \cup \{\langle\langle h, s\rangle, hash(destination)\rangle\}$
rof
;
if $nullable(RE_j) \rightarrow$
{ The final states all have a right language containing the empty string }
$F := F \cup \{h\}$
[] $\neg nullable(RE_j) \rightarrow$ **skip**
fi
od;
return $\langle Q, \delta, hash(RE_{init}), F\rangle$
cnuf

the resulting DFA, since the algorithm removes its only in-transition. At the implementation level, this would mean that an amount of total storage used for the transition function would be wastefully occupied. In our implementation of Algorithm 3, the removal of such dead states and associated transitions is quite simple, but implementation-dependent. For this reason, and for the sake of brevity, these details have been omitted from Algorithm 3.

5 Characterising hash functions

5.1 A basis for measuring hash function quality

It has been established above that the automaton constructed through state merging is a super-automaton of the language associated with the algorithm's input regular expression. Informally, one might say that the "difference" between the languages recognised by these respective automata reflects the quality of the hash function used to generate the super-automaton. In this section, we propose a more precise

and formal notion of the quality of the hash function. In the penultimate section, preliminary empirical investigations to assess these ideas are described.

The notion of hash function quality is based on the perception that the language associated with the algorithm's input regular expression has a unique minimum DFA, which may or may not be produced by Algorithm 3. The approach to finding this unique minimum DFA described in [6] is based on the notion of k-equivalence between states. The k-equivalence relation between states is inductively defined as follows [2]:

Definition 6. *Two states t_1 and t_2 are:*

- *0-equivalent iff they are both either accepting or rejecting states.*
- *k-equivalent iff for all input symbols s on the states t_1 and t_2, the next states $\delta(t_1, s)$ and $\delta(t_2, s)$ are $(k-1)$-equivalent.*
- **-equivalent iff they are k-equivalent for all values of k larger than some constant value.*

In [5] it is shown that if, in an automaton that contains $|Q|$ states, two states are k-equivalent for $k = |Q| - 2$, then these two states are *-equivalent. If two states are *-equivalent, it means that they represent the same regular language.

In the above-mentioned minimization algorithm, pairs of states of a given DFA are examined to determine their k-equivalence status. The algorithm therefore implicitly determines membership of k-equivalence classes in general, and of *-equivalence classes in particular. States in each *-equivalent class are eventually merged into a single state, resulting in the required unique minimal FA.

The foregoing suggests an approach to measuring the quality of a hash function, namely to associate quality with the extent to which state merging (caused by hash-clashes) approximates the merging of *-equivalent states.

However, before formalising this insight, note that hash functions in Algorithm 3 take regular expressions as input. For this reason, the notion k-equivalent regular expressions needs to be defined. The definition is analogous to the definition of k-equivalent states, namely:

Definition 7. *Two regular expressions RE_i and RE_j are:*

- *0-equivalent iff $nullable(RE_i) = nullable(RE_j)$*
- *k-equivalent iff $first(RE_i) = first(RE_j)$ and for all $s \in first(RE_i)$, $s^{-1}RE_i$ and $s^{-1}RE_j$ are $(k-1)$-equivalent.*

When a given hash function maps two regular expressions to the same value, we may enquire about the maximum k-equivalence. They may not have any equivalence relationship at all, or they be maximally k-equivalent for some $k < |Q|-2$, or they may be *-equivalent. Various weighting schemes could be proposed to reflect the overall quality of all clashes during a run of Algorithm 3: the higher the maximal k-equivalent status of two clashing regular expressions, the more favourably the clash should weigh in the overall measure. In the preliminary empirical experiments discussed below, the percentage of hash clashes that result from *-equivalent regular expressions relative to the total number of hash clashes is used as a measure of the quality of a number of different hash functions. Other measures of quality are not considered at this stage.

5.2 Ideal hash functions

The foregoing raises the question: what are the characteristics of an ideal hash function? To give such a characterisation, note that the signature of hash functions in Algorithm 3 are of the form $h : R^e \to \mathbb{N}$, where R^e is the set of regular expressions. Suppose that R^ℓ is the set of regular languages, and that $L : R^e \to R^\ell$, so that $L(R)$ is the regular language associated with regular expression R. Finally, suppose that f denotes a function $f : R^\ell \to \mathbb{N}$. An ideal hash function may now be defined as the composition of the latter two functions as follows:

Definition 8. *h is an ideal hash function for R^e iff $h = f \cdot L$ and f is an injection.*

This means an ideal hash function maps all regular expressions that have the same language (and only those expressions) to the same natural number. Put differently, an ideal hash function maps regular expressions to the same value if and only if they are *-equivalent. Thus, if it were possible to find an ideal hash function for use in Algorithm 3, then the algorithm would be guaranteed to produce the *minimum exact* DFA for the input regular expression.

Note that our notion of an *ideal* hash function for regular expressions is slightly different from the conventional notion of a *perfect* hash function for a set. The latter is an injection from that set to the natural numbers. In fact, the above definition could be reformulated to indicate that h is an ideal hash function on R^e if f is a perfect hash function on R^ℓ.

In fact, the remapping in Algorithm 1 may be viewed abstractly as the application of a perfect hash function on R^e, not to regular expressions representing the right language of a state, but to regular expressions representing the *language of a state*. This is the concept next defined.

5.3 A regular expression for the language of a state

Consider a regular expression, R, that is to be hashed or remapped in one of the algorithms previously discussed. Its language, $L(R)$, corresponds with the so-called *right* language of its associated state in the constructed FA, denote by $\overrightarrow{L}(state(R))$. That same state also has a *left* language, $\overleftarrow{L}(state(R))$, which is the set of all prefixes of all strings of the FA that pass through $state(R)$. Indeed the set of all strings passing through $state(R)$, is is called the language of $state(R)$. It is denoted by $L(state(R))$, and is given by $\overleftarrow{L}(state(R)) \cdot \overrightarrow{L}(state(R))$.

However, when constructing $state(R)$ during the execution of any of the algorithms, a regular expression whose language is $\overleftarrow{L}(state(R))$ is not available. All that is available during any iteration is the initial regular expression, RE_{init}, and the regular expression currently under consideration, R. Fortunately, this information is sufficient to find an explicit form for a regular expression, R', whose language corresponds $L(state(R))$, namely:

$$R' = (\Sigma^* \cdot R) \cap RE_{init} \tag{1}$$

Note that $(\Sigma^* \cdot R)$ designates all possible strings that end in a string that is in R's right language. Intersecting these strings with RE_{init} ensures that only strings in $L(state(R))$ remain.

Thus, abstractly, the remapping in Algorithm 1 can be viewed as a perfect hash function that is applied, not to the right language of a regular expression R, but to the corresponding regular expression R' as defined abovince the language of each state in a DFA is unique, this perfect hash function never merges any states. Of course, this claim has to be qualified in terms of the precise way in which a given application implements the remapping in the algorithm. A given implementation might have a more liberal notion of regular expression equality than strict lexicographic equality, taking into account operator properties such as idempotence or commutativity. (For example a may be treated as the same regular expression as $a \cup a$, and/or $a \cup b$ as the same regular expression as $b \cup a$, etc.)

Below we report briefly on a preliminary experiment in which Algorithm 2 is run with a variety of hash functions that are applied to R, while Algorithm 3 is run with these same hash functions applied to R'.

5.4 The effects of the modulo function

Most hash functions are of the form $(h(r)$ **mod** $n)$—i.e. they they apply modulo n to some integer value that they compute, where n reflects the address space being hashed to. This application of modulo n will clearly tend to undermine the quality of hash function h. If h happened to be an ideal hash function, then there is no guarantee that $(h(r)$ **mod** $n)$ will deliver ideal behaviour.

6 Preliminary empirical investigations

In preliminary experiments to test the above ideas, Gödel numbers [4] were used to generate over 190 000 short different regular expressions of length 7, based on 4-character alphabet. Algorithm 1 was applied to each of these regular expressions. The largest FA generated by Algorithm 1 had 7 states.

In order to apply Algorithms 2 and 3, various hash functions were selected, based on the recommendations in [7]. In particular, various morphisms were selected, each structurally mapping the regular expressions to a different hash function. By this we mean that wherever an operator occurs in a regular expression, a corresponding integer operator is selected in the hash function which is structurally similar to the regular expression operator. For example, since the regular expression $\cup$-operator is idempotent and commutative, a hash function should be used that relies on an integer operator with these properties wherever the $\cup$-operator occurs in the regular expression. Such an integer operator might be, for example, addition, bitwise-and, bitwise-or, etc. Eight such mappings are shown in table 1.

Algorithm 2 was repeatedly invoked to construct DFAs for each of the more than 190000 regular expressions. Each of the eight hash functions in table 1 was used, as well as modulo i $(i = 2, \ldots, 5)$ variants of the each hash function. Thus $8 \times 5 = 40$ DFAs were constructed for each regular expression. This entire experiment was then repeated using a modified version of Algorithm 3 in which the regular expression in equation (1) was used to determine the hashed value of each state, instead of the derivative representing the state's right language. (The results of Algorithms 2 and 3 based on hashing the right language of a state were similar, and consequently, the results of Algorithm 3 run in this mode are not further discussed here.)

In each application of Algorithms 2 and 3, whenever two states were to be merged, their *-equivalent status was assessed. This was done by verifying the k-equivalent

status of the two states up to $k = |Q| - 2$, the point at which *-equivalence is attained [5]. Since the exact value of $|Q|$ was not known *a priori*, the upper bound on $|Q|$ established by Algorithm 1 was used, namely 7.

Mapping 1	Mapping 2
$\emptyset \mapsto 000\ldots000_{16}$	$\emptyset \mapsto 000\ldots000_{16}$
$\varepsilon \mapsto 000\ldots000_{16}$	$\varepsilon \mapsto 000\ldots000_{16}$
$arg^? \mapsto 000\ldots000_{16} \wedge arg$	$arg^? \mapsto 000\ldots000_{16} \wedge arg$
$arg^+ \mapsto \neg arg \vee (arg \vee (1 << (n-1)))$	$arg^+ = \neg arg \vee (arg \vee (1 << (n-1)))$
$arg^* \mapsto arg \vee (1 << (n-1))$	$arg^* = arg \vee (1 << (n-1))$
$arg_1 \cap arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cap arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cup arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cup arg_2 \mapsto arg_1 \wedge arg_2$
$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$	$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$
Mapping 3	Mapping 4
$\emptyset \mapsto 000\ldots000_{16}$	$\emptyset \mapsto 000\ldots000_{16}$
$\varepsilon \mapsto 000\ldots000_{16}$	$\varepsilon \mapsto 000\ldots000_{16}$
$arg^? \mapsto 000\ldots000_{16} \vee arg$	$arg^? \mapsto 000\ldots000_{16} \vee arg$
$arg^+ \mapsto \neg arg \vee (arg \vee (1 << (n-1)))$	$arg^+ \mapsto \neg arg \vee (arg \vee (1 << (n-1)))$
$arg^* \mapsto arg \vee (1 << (n-1))$	$arg^* \mapsto arg \vee (1 << (n-1))$
$arg_1 \cap arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cap arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cup arg_2 \mapsto arg_1 \vee arg_2$	$arg_1 \cup arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$	$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$
Mapping 5	Mapping 6
$\emptyset \mapsto FFF\ldots FFF_{16}$	$\emptyset \mapsto FFF\ldots FFF_{16}$
$\varepsilon \mapsto 000\ldots000_{16}$	$\varepsilon \mapsto 000\ldots000_{16}$
$arg^? \mapsto 000\ldots000_{16} \wedge arg$	$arg^? \mapsto 000\ldots000_{16} \wedge arg$
$arg^+ \neg arg \vee (arg \vee (1 << (n-1)))$	$arg^+ \neg arg \vee (arg \vee (1 << (n-1)))$
$arg^* \mapsto arg \vee (1 << (n-1))$	$arg^* \mapsto arg \vee (1 << (n-1))$
$arg_1 \cap arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cap arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cup arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cup arg_2 \mapsto arg_1 \wedge arg_2$
$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$	$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$
Mapping 7	Mapping 8
$\emptyset \mapsto FFF\ldots FFF_{16}$	$\emptyset \mapsto FFF\ldots FFF_{16}$
$\varepsilon \mapsto 000\ldots000_{16}$	$\varepsilon \mapsto 000\ldots000_{16}$
$arg^? \mapsto 000\ldots000_{16} \vee arg$	$arg^? \mapsto 000\ldots000_{16} \vee arg$
$arg^+ \neg arg \vee (arg \vee (1 << (n-1)))$	$arg^+ \neg arg \vee (arg \vee (1 << (n-1)))$
$arg^* \mapsto arg \vee (1 << (n-1))$	$arg^* \mapsto arg \vee (1 << (n-1))$
$arg_1 \cap arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cap arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cup arg_2 \mapsto arg_1 \vee arg_2$	$arg_1 \cup arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$	$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$

Table 1. Mappings of regular expression operators to hash function operators

Tables 2 and 3 summarise the results of these experiments. Columns headed *-eq % indicate the percentage of hash clashes (and thus merged states) that turned out to be *-equivalent for the specific hash function version. Columns headed "States" indicate the size of the largest DFAs (in terms of number of states) generated by the respective hash function. The tables reveal the following patterns:

– The relative hash function performance (as indicated in the *-eq% columns) appears very similar for the two algorithms: good hash functions seem to perform consistently well, and bad functions consistently badly. In fact, Spearman's rank correlation test was applied to the *-equivalent rankings in the two tables of the

	No MOD		MOD 2		MOD 3		MOD 4		MOD 5	
Mapping	*-eq %	States	*-eq %	States	*-eq %	States	*-eq %	States	*-eq %	States
1	77	4	73	2	73	3	75	4	75	4
2	79	4	73	2	74	3	76	4	76	4
3	83	4	75	2	76	3	78	4	79	4
4	82	4	74	2	75	3	77	4	77	4
5	77	4	73	2	73	3	75	4	75	4
6	79	4	73	2	74	3	76	4	75	4
7	83	4	75	2	76	3	78	4	79	4
8	81	4	74	2	74	3	76	4	76	4

Table 2. Algorithm 2 results: hashed regular expressions represent right language of states.

	No MOD		MOD 2		MOD 3		MOD 4		MOD 5	
Mapping	*-eq %	States	*-eq %	States	*-eq %	States	*-eq %	States	*-eq %	States
1	69	1	69	1	69	1	69	1	69	1
2	71	2	69	1	71	2	69	1	71	2
3	80	4	69	1	71	2	69	1	71	2
4	71	3	74	2	71	2	69	1	71	2
5	69	1	69	1	69	1	69	1	69	1
6	69	1	69	1	69	1	69	1	69	1
7	80	4	74	2	74	3	76	4	77	4
8	71	3	69	2	70	3	76	4	77	4

Table 3. Algorithm 3 results: hashed regular expressions represent full language of states.

hash functions without the modulo operation. A correlation value of 0.92 was obtained, which is well above the 95% confidence level of 0.72 (for sample size of 8) for rejecting the hypothesis that the hash function performance differed in the two algorithms.

- Data in table 2 indicating the largest DFA generated under each hash function is very much in line with expectations. All hash functions of a given modulo produce the same size largest DFAs, and these largest sizes rise as the modulo value rises. They attain a maximum size of 4, when modulo 4 is reached. This suggests (but does not prove) that 4 is the maximum size of the minimized DFA generated from the more than 190000 regular expressions.
- By contrast, data in table 3 in relation to the largest DFA generated under various hash functions, does not seem to be influenced significantly by the modulo operation. In fact, the overall quality measures in this table are lower than in table 2. If the inference above is correct that the maximum size of the minimized DFA is 4, then maximum DFA sizes increasingly less than 4 lead to super-automata increasingly different from the associated exact automata. This suggests that hashing on the language of a state is not a good idea. The reasons for this relatively poor performance will be further researched in future work.
- Entries in the *-eq % columns seem surprisingly high. It is interesting to note in table 3 that all hash functions that reduce the maximum sized DFA (and therefore all DFAs) down to 1 state, have a *-equivalent rating of 69%. This means that the percentage *-equivalent mergers attributable to *different* regular expressions being

hashed to the same value, ranges across the two tables from $69\% - 69\% = 0\%$ to a maximum of $83\% - 69\% = 14\%$.
- Hash functions 3 and 7 appear to perform consistently well, whereas hash functions 1 and 5 (and possibly 6) perform consistently badly. It would seem that the last three mappings in each block of table 1 play a critical role in determining the hash function quality. Worst case behaviour arises when the regular expression operator $\cup$ is associated with the bitwise operator $\wedge$ (see mappings 1, 2, 5 and 6), which significantly improves when the association is switched to the bitwise operator $\vee$ (see mappings 3, 4, 7 and 8). Optimal performance is reached (in mapping 3 and 7) when, in addition, $\cap$ is mapped to $\wedge$, but this mapping also leads to worst case performance is also reached (in mappings 1 and 5) if $\cup$ is wrongly mapped.

Even though the automata constructed are small in size, a large range of regular expressions have been tested, representing a diverse range of regular expression structures. Nevertheless, a shortcoming of this approach is it becomes impractical for larger regular expressions: the Gödel numbers involved become extremely large, making it impossible to iterate over them.

In the future, we intend generating a sample of random large regular expressions, to assess the impact of different hash functions under such circumstances.

7 Conclusions and Further Work

In this article, the consequences of regular expression hashing as a means of finite state automaton reduction was explored based on variations of Brzozowski's Algorithm. It was shown that a super-automaton is always constructed, no matter what the hash function may be. It was also demonstrated that a non-deterministic automaton can be constructed, and a new algorithm was put forward for constructing a deterministic FA, using the same approach as the original two algorithms.

An approach was proposed to measuring the quality of a hash function that derives a super-automaton of an exact automaton, based on k-equivalence classes on regular expressions. A derivation was also presented to represent the language of a state with regular expressions. These ideas were empirically tested on a large sample of relatively small regular expressions and their associated automata.

Further work will focus on searching for hash functions that are closer to the ideal, and on gaining a more precise understanding of why some hash functions are better than others, given the k-equivalence criteria and the definition of an ideal hash function. This will include exploring substitution variations on integer functions for regular expression operators.

References

1. J. Brzozowski: *Derivatives of regular expressions.* Journal of the Association of Computing Machinery, 11 October 1964, pp. pages 481–494.
2. S. Epp: *Discrete Mathematics with Applications*, International Tomson Publishing, Inc., 1995.
3. M. Frishert: *FIRE Works & FIRE Station: A finite automata & regular expression playground*, tech. rep., Technical University Eindhoven, 2004.
4. R. McNaughton: *Elementary Computability, Formal Languages and Automata*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
5. B. W. Watson: *A taxonomy of finite automata minimization algorithms*, tech. rep., Technical University Eindhoven, 1994.

6. B. W. Watson and J. Daciuk: *An efficient incremental DFA minimization algorithm.* Journal of Natural Language Engineering, 9(1) March 2003, pp. 49–64.
7. B. W. Watson, D. G. Kourie, E. Ketcha Ngassam, T. Strauss, and L. Cleophas: *Efficient automata constructions and approximate automata.* International Journal of Foundations of Computer Science, Vol. 19(1) 2008, pp. 185–193.