International Journal of Foundations of Computer Science
Vol. 19, No. 1 (2008) 185–193
© World Scientific Publishing Company

EFFICIENT AUTOMATA CONSTRUCTIONS AND APPROXIMATE AUTOMATA[*]

BRUCE W. WATSON[†]
FASTAR Research group
Department of Computer Science, University of Pretoria
Pretoria, 0002, South Africa
FST Labs Inc., Kelowna, Canada

DERRICK G. KOURIE, TINUS STRAUSS
Department of Computer Science, University of Pretoria
Pretoria, 0002, South Africa

ERNEST KETCHA
School of Computing, University of South Africa
Pretoria, 0002, South Africa

and

LOEK CLEOPHAS
Faculty of Computing Science and Mathematics
Eindhoven University of Technology, Eindhoven, Netherlands

Received 8 November 2007
Accepted 14 February 2007
Communicated by J. Holub

In this paper, we present data structures and algorithms for efficiently constructing *approximate automata*. An approximate automaton for a regular language L is one which accepts *at least* L. Such automata can be used in a variety of practical applications, including network security pattern matching, in which false-matches are only a performance nuisance. The construction algorithm is particularly efficient, and is tunable to yield more or less *exact* automata.

Keywords: Automaton construction; approximate automata; memory efficiency; hash functions.

[*]This paper is a corrected version of the one appearing in the *Proceedings of the Prague Stringology Conference 2006*. In the original paper, an incorrect algorithm was given for constructing a deterministic approximate automaton.
[†]bruce@bruce-watson.com, bruce@watson.name

1. Introduction

In this paper, we present data structures and algorithms for efficiently constructing an 'approximate automaton' from a regular expression. Very large automata are finding application in areas as diverse as computational linguistic, network intrusion detection, text indexing, and silicon chip design. These problem domains intrinsically have very large amounts of data—both in the form of input strings being processed by finite automata, and also in the size of the automata themselves.

A great deal of effort has been invested in tuning algorithms for processing a string for acceptance by an automaton, or for pattern matching using the automaton[a]. Recently, much less research and implementation effort has been devoted to the efficiency during construction of very large automata[b]. In some of the newer application domains, the 'exactness' of the automata is proving to be less of an issue than the performance of the algorithms constructing and using the automata. In particular, in network intrusion detection, a pattern matching finite automaton may accept some 'extra' words without ill effects—the algorithm merely detects an additional matched pattern (corresponding to a network security attack), which is subsequently discarded during further vetting of the pattern[c]. In such an application, we can use an approximate automaton—one which accepts the intended language and perhaps some additional strings. Approximate automata have also been derived for a field of pattern matching, where they are known as *factor oracles* [1, 3].

Section 2 gives a definition of the problem, along with discussion of some of the existing solutions. Section 3 gives the new algorithm, while Section 4 discusses choices of hash functions. Finally, Section 5 gives some discussion points, conclusions and future work. Throughout this paper, we use standard definitions of finite automata, which are not discussed or defined further in detail.

2. Problem Statement

In this section, we give a brief overview of the problem and existing work on solutions. In Algorithm 2.1, we begin with Brzozowski's automata construction algorithm [2, 10]. The algorithm takes as input a regular expression over alphabet Σ and directly produces a finite automaton. (Note that the **skip** statement does nothing — it is included for completeness so that the **if-fi** statement has two complementary branches. For regular expression E and alphabet symbol a, the expression $a^{-1}E$ denotes the left-derivative of E with respect to a; for more on such derivatives, see [2].) (This algorithm is arguably one of the simplest and most elegant algorithms, although it is not always efficiently implemented, giving the incorrect impression that some of the bitvector-based algorithms (such as those of

[a]Cf. the proceedings of conferences such as *Prague Stringology Workshop*, *Conference on Implementations and Applications of Automata* and *Combinatorial Pattern Matching*.

[b]In the last decades, asymptotic improvements were made by Champarnaud, Ponty, Ziadi, Chang, Paige, Antimirov, and Watson, among others.

[c]Such further vetting is characteristic of intrusion detection, in which network traffic is rapidly scanned for patterns; pattern 'hits' are subsequently examined for further characteristics before classifying them as a real network security attack.

Algorithm 2.1 (Brzozowski's construction):

```
func $Brzconstr(E) \rightarrow$
    $Q, \delta, F := \emptyset, \emptyset, \emptyset$;
    $done, todo := \emptyset, \{E\}$;
    do $todo \neq \emptyset \rightarrow$
        let $p$ be some state such that $p \in todo$;
        $done, todo := done \cup \{p\}, todo \setminus \{p\}$;
        $Q := Q \cup \{p\}$;
        { build out-transitions from $p$ on all alphabet symbols }
        for $a : \Sigma \rightarrow$
            { compute the left derivative of $p$ with respect to $a$ }
            $destination := a^{-1}p$;
            if $destination \notin done \cup todo \rightarrow$
                { $destination$'s out-transitions are still to be built }
                $todo := todo \cup \{destination\}$
            ▯ $destination \in done \cup todo \rightarrow$ skip
            fi;
            { make a transition from $p$ to $destination$ on $a$ }
            $\delta(p, a) := destination$
        rof;
        { if $p$ is nullable, make it a final state }
        if $\varepsilon \in \mathcal{L}(p) \rightarrow$
            { $p$ should be a final state }
            $F := F \cup \{p\}$
        ▯ $\varepsilon \notin \mathcal{L}(p) \rightarrow$ skip
        fi
    od;
    { lang. of autom. $(Q, \Sigma, \delta, E, F)$ = lang. of reg. expr. $E$ }
    return $(Q, \Sigma, \delta, E, F)$
cnuf
```

□

Glushkov, McNaughton-Yamada, Berry-Sethi, among others) are intrinsically more efficient.)

The abstract states in Algorithm 2.1 have 'internal structure', meaning that they are in fact regular expressions. In practice, the regular expressions are mapped on-the-fly to integers to store the transition function δ in a lookup table and final state set F as a bitvector (indexed by state). That gives Algorithm 2.2, in which state set Q is replaced by a set of integers, the start state is state zero, the signature of δ is appropriately changed, and a 'remapping' data structure is used to map the abstract states to integers. In this algorithm, an abstract state is remapped (assigned an integer representation) when it is first encountered/created (as opposed to when its out-transitions are constructed), since the integer representation may be needed as a transition destination immediately.

The worst-case running time of this algorithm (indeed, of all deterministic finite

Algorithm 2.2 (Brzozowski's construction with remapping):

```
func $Brzconstr'(E) \rightarrow$
    $next, \delta, F, remap := 0, \emptyset, \emptyset, \emptyset;$
    $remap[E], next := next, next + 1;$
    $done, todo := \emptyset, \{E\};$
    do $todo \neq \emptyset \rightarrow$
        let $p$ be some state such that $p \in todo$;
        $done, todo := done \cup \{p\}, todo \setminus \{p\};$
        { build out-transitions from $p$ on all alphabet symbols }
        for $a : \Sigma \rightarrow$
            { compute the left derivative of $p$ with respect to $a$ }
            $destination := a^{-1}p;$
            if $destination \notin done \cup todo \rightarrow$
                { $destination$'s out-transitions are still to be built }
                $todo := todo \cup \{destination\};$
                { give $destination$ an integer representation now though }
                $remap[destination], next := next, next + 1$
            ‖ $destination \in done \cup todo \rightarrow$ skip
            fi;
            { make a transition from $p$ to $destination$ on $a$ }
            $\delta(remap[p], a) := remap[destination]$
        rof;
        { if $p$ is nullable, make it a final state }
        if $\varepsilon \in \mathcal{L}(p) \rightarrow$
            { $remap[p]$ should be a final state }
            $F := F \cup \{remap[p]\}$
        ‖ $\varepsilon \notin \mathcal{L}(p) \rightarrow$ skip
        fi
    od;
    { lang. of autom. $(\{0, \ldots, next - 1\}, \Sigma, \delta, 0, F)$ = lang. of reg. expr. $E$ }
    return $(\{0, \ldots, next - 1\}, \Sigma, \delta, 0, F)$
cnuf
```

□

automata construction algorithms) is exponential in the size of the regular expression. However, we are more concerned with those parts of the algorithm and data structures which are tunable.

We make the following observations about Brzozowski's construction algorithm[d]:

1. The sets *done* and *todo* both contain abstract states, whereas they could just contain the integer representations of states, while using the inverse of *remap* to recover the abstract state (which is needed in building the out-transitions). We do not discuss this optimization further, as it is already used in most implementations.

2. In pathological examples, the *todo* set (containing abstract states that still need to be constructed) can grow at one time during construction to contain all states which will ever be built (except for the current state p). The only solutions to this problem are domain-specific; that is, the size of *todo* is sometimes bounded by representing *todo* as a queue or as a stack (yielding, respectively, a depth-first or a breadth-first construction of the automaton's transition graph).

3. The performance of the algorithm depends heavily on the quality of the *remap* representation for fast lookups. There are numerous efficient implementations for *remap*, including hash tables, balanced trees, etc. This is not discussed further here.

4. During construction, the remapper (data structure *remap*) grows to include a mapping from *all* abstract states to their respective integer representations. The memory consumed by *remap* is therefore significant; worse-still, it is only freed after the entire automaton is constructed.

The most promising area for improvement is the last point, for which the following solutions exist:

1. [8, 6, 7, 5] give a space-efficient data structure combining representations of regular expression E, all of the derivative regular expressions in *remap* (the set of states) and the automaton itself.

2. A reachability-based algorithm was presented in [11, 12]. That algorithm limits *remap* to those states which are reachable from states still in *todo*.

We now turn to the new algorithm, which is combinable with these two pre-existing solutions.

3. New Algorithm

One implementation of *remap* uses a hash table with hash function h, which maps regular expressions (the abstract states) to integers. In the event that two regular

[d]Some of these observations were previously made in [11, 12]—though about another construction algorithm.

Algorithm 3.1 (New hash-based construction):

```
func $Brzconstr''(E) \rightarrow$
    $Q, \delta, F, remap := \emptyset, \emptyset, \emptyset, \emptyset;$
    $done, todo := \emptyset, \{E\};$
    do $todo \neq \emptyset \rightarrow$
        let $p$ be some state such that $p \in todo$;
        $done, todo := done \cup \{p\}, todo \setminus \{p\};$
        $Q := Q \cup \{h(p)\};$
        { build out-transitions from $p$ on all alphabet symbols }
        for $a : \Sigma \rightarrow$
            { compute the left derivative of $p$ with respect to $a$ }
            $destination := a^{-1}p;$
            if $destination \notin done \cup todo \rightarrow$
                { $destination$'s out-transitions are still to be built }
                $todo := todo \cup \{destination\}$
            [] $destination \in done \cup todo \rightarrow$ skip
            fi;
            { make a transition from $h(p)$ to $h(destination)$ on $a$ }
            $\delta(h(p), a) := \delta(h(p), a) \cup \{h(destination)\}$
        rof;
        { if $p$ is nullable, make it a final state }
        if $\varepsilon \in \mathcal{L}(p) \rightarrow$
            { $h(p)$ should be a final state }
            $F := F \cup \{h(p)\}$
        [] $\varepsilon \notin \mathcal{L}(p) \rightarrow$ skip
        fi
    od;
    { lang. of autom. $(Q, \Sigma, \delta, h(E), F)$ = lang. of reg. expr. $E$ }
    return $(Q, \Sigma, \delta, h(E), F)$
cnuf
```

□

expressions hash-collide, a mechanism is normally used to check whether the regular expressions are indeed identical—typically using 'hash-buckets', rehashing, etc. (as is found in elementary data structure textbooks such as [4]). Unfortunately, all of those collision-resolution mechanisms involve representing the abstract states themselves. Our new algorithm eliminates this, thereby allowing hash collisions: two abstract states which hash to the same value are simply mapped to the same integer state. Indeed, the hashed values can be directly used as the states, as in Algorithm 3.1 where we reintroduce state set Q; state sets Q and F can now be implemented as sets of integers. Here, two hashed-together states may have out-transitions to states which do not hash together, thus producing a nondeterministic automaton; this is reflected in the new assignment updating δ — it is now a nondeterministic transition function.

4. Hash Functions and Their Implications

The primary difference between a normal automaton construction algorithm (such as Brzozowski's algorithm) and Algorithm 3.1 is that the latter will merge two states whenever there is a hash collision (possibly yielding a nondeterministic automaton); such merging would not otherwise have occurred in any of the standard construction algorithms. The resulting automaton is an *approximate automaton* as it will accept additional words not in the language of regular expression E. Precisely how often additional words are accepted clearly depends on how often hash collisions occur, which depends in turn on the hash function h. The approximate automaton can be forced arbitrarily close to an *exact* automaton for E by increasing the number of bits in the range of h and making h appropriately more intricate[e].

While the design of h is crucial, our ongoing experiments give these guiding requirements:

- It should be structurally inductive on regular expressions.
- It should map to unsigned integers.
- The atomic 'letter' (single symbol) regular expressions should map to the character itself, e.g. $h(a) = \mathit{unsigned}(a)$.
- The remaining two atomic regular expressions (empty string and empty set) should map to infrequently or unused characters, such as -1 and -2.
- The hash of Kleene closure (the star operator) should set a high-bit in the hash of the star's operand's hash, e.g. $h(F^*) = h(F)\&(1 << (\mathit{numbits}-1))$. In this case, the hash function can also be designed for idempotence of Kleene closure, i.e. $h(F^{**}) = h(F^*)$, etc. Such 'design-for-identities' allows us to specifically hash-collide states which look dissimilar, but are equivalent, thereby achieving a measure of *minimization*[f].
- The hash of a union/or regular expression should combine the two sub-hashes via an associative and commutative operator, such as exclusive-or.
- The hash of two concatenated regular expressions should combine the two sub-hashes while simultaneously being anti-symmetrical (as concatenation is), such as bitwise concatenation, or left-shifting the first sub-hash before exclusive-oring with the second sub-hash.

In short, the hash function should reflect the algebraic properties of the regular operators themselves.

Considering the update of transition function δ in Algorithm 3.1, we could additionally restrict the hash function such that we always obtain a *deterministic* automaton. The restriction is easily derived from the fact that δ must then be a

[e]When the number of bits is equal to the largest derivative of E, h can be used to directly encode each such derivative regular expression—giving perfect hashing.

[f]Other regular identities which may be specifically hash-collidable include $F \cdot \emptyset = \emptyset$, $F \cdot \varepsilon = F$, $\varepsilon^* = \varepsilon$, etc.

singleton function, mapping state p and alphabet symbol a to a *single* state. The structural restriction is (for all states p, q and alphabet symbols a):

$$h(p) = h(q) \Rightarrow h(a^{-1}p) = h(a^{-1}q)$$

It is not yet clear how easily we can derive hash functions satisfying this requirement.

Interestingly, the use of a hash function to generate the state-set enables us to *a priori* choose the number of states in the final automaton. Rather than accumulating the hashed values in variable Q, we initially fix our state set as $\{0, \ldots, n\}$ (for some n). Subsequently, we always use $h(p) \mod n$ instead of $h(p)$, thereby bringing all states into the appropriate range. This can be particularly useful in cases where dynamic memory (re)allocation is costly while building the automaton.

5. Discussion and Future Work

We have presented an efficient new algorithm for constructing approximate finite automata. The 'approximateness' of the finite automata (that is, how few 'extra' words they accept over and above their intended language) can be controlled by choice of a hash function, some guidelines for which were presented. Unlike other automata construction algorithms, the number of states can be fixed *a priori*.

There remain a number of interesting research questions and tasks:

1. The new algorithm and a variety of hash functions should be benchmarked; this work is ongoing.
2. The effects of various hash functions should be tested in practice (for example, in intrusion detection) for how close the resulting automaton is to the desired language.
3. Some operations in the new algorithm can be parallelized. A parallelization of Brzozowki's algorithm is the subject of another paper in this journal issue. It would be interesting to know whether further parallelization opportunities exist in the new algorithm.
4. In data structure design, the size of the hash tables are typically chosen to be a prime number (and therefore the hash key is reduced modulo this size), as this reduces the probability of collisions. It would be interesting to know whether a similar property exists in the new algorithm, with typical choices of hash functions.

References

[1] C. Allauzen, M. Crochemore, and M. Raffinot: "Factor oracle: A new structure for pattern matching", in *Proceedings of SOFSEM*, 1999, pp. 295–310.

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

[3] L. Cleophas, B. W. Watson, and G. Zwaan: "Constructing factor oracles", in *Proceedings of the Eighth Prague Stringologic Conference*, J. Holub, ed., Prague, Czech Republic, Sept. 2003, Czech Technical University.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: *Introduction to Algorithms*, The MIT Press, second ed., 2001.

[5] M. Frishert: "FIRE Works & FIRE Station: A finite automata and regular expression playground", Master's thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 2005.

[6] M. Frishert, L. Cleophas, and B. W. Watson: "FIRE station: An environment for manipulating finite automata and regular expression views", in Salomaa [9].

[7] M. Frishert and B. W. Watson: "Combining regular expressions with (near-)optimal Brzozowski automata", in Salomaa [9].

[8] M. Frishert, B. W. Watson, and L. Cleophas: "The effects of rewriting regular expressions on their accepting automata", in *Proceedings of the Eighth Conference on Implementations and Applications of Automata*, O. Ibarra and D. Zhu, eds., Santa Barbara, USA, July 2003, Springer-Verlag, pp. 304–305.

[9] K. Salomaa, ed., *Proceedings of the Ninth Conference on Implementations and Applications of Automata*, Kingston, Canada, July 2004, Springer-Verlag.

[10] B. W. Watson: "Taxonomies and Toolkits of Regular Language Algorithms", PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, Sept. 1995.

[11] ———: "An early-retirement plan for the states", in *Proceedings of the Third Prague Stringologic Workshop*, J. Holub, ed., Prague, Czech Republic, Sept. 1998, Czech Technical University, pp. 119–124.

[12] ———: "Reducing memory requirements during finite automata construction". *Software — Practice & Experience*, 34(3) 2004, pp. 239–248.