Failure Deterministic Finite Automata

Derrick G. Kourie[1], Bruce W. Watson[2], Loek Cleophas[1,3], and Fritz Venter[1]

[1] University of Pretoria
[2] Stellenbosch University
[3] Eindhoven University of Technology
{dkourie,bruce,loek,fritz}@fastar.org

Abstract. Inspired by failure functions found in classical pattern matching algorithms, a failure deterministic finite automaton (FDFA) is defined as a formalism to recognise a regular language. An algorithm, based on formal concept analysis, is proposed for deriving from a given deterministic finite automaton (DFA) a language-equivalent FDFA. The FDFA's transition diagram has fewer arcs than that of the DFA. A small modification to the classical DFA's algorithm for recognising language elements yields a corresponding algorithm for an FDFA.

Keywords: failure arcs, DFA, formal concept analysis

1 Introduction

It is well-known that there is a mapping between deterministic finite automata (DFAs) and regular languages. Let $\mathcal{L}(\mathcal{D}) \subseteq \Sigma^*$ denote the regular language associated with DFA $\mathcal{D}$, the DFA being defined on an alphabet Σ and having δ as its transition function. The transition function maps a state / symbol pair to a new state, i.e. $\delta(q, a) = p$ where $q, p \in Q$, the DFA's set of states, and $a \in \Sigma$.

Given an arbitrary finite-length string $x \in \Sigma^*$, there is a classical algorithm to test whether $x \in \mathcal{L}(\mathcal{D})$. The algorithm uses δ to transition from state to state as it processes x on a character by character basis. It starts from the DFA's start state and terminates once all characters in x have been processed. Only if a final state has been reached at termination does the algorithm affirm that $x \in \mathcal{L}(\mathcal{D})$.

Such an algorithm takes time $\mathcal{O}(|x|)$, assuming that $\delta(q, a)$ is computed in constant time. It uses $\mathcal{O}(|\Sigma| \times |Q|)$ space, as δ has to be stored. Applications of the algorithm vary widely, with the underlying DFA possibly involving millions of states and transitions. Consequently, research efforts have been directed at improving on the algorithm's space or time efficiency. Examples include DFA minimisation [22], hard-coding and cache manipulation [10], various automata transformations [6,3], various strategies for storing sparse matrices [21,8], and other strategies to reduce representation sizes [5].

Here we focus on improving on space efficiency by relying on *failure* DFAs (FDFAs). The formalism derives from the failure functions found in classical pattern matching algorithms [1,11,4]. Recall, for example, the Aho-Corasick algorithm [1] which takes a finite set of patterns and identifies all their occurrences in a text. One version uses a DFA, while a second version uses a trie DFA [9] with a so-called failure function. The latter version removes arcs that do not contribute to the definition of patterns, replacing them judiciously with arcs derived from a failure function. The result is a trie DFA, decorated by various failure arcs. The standard acceptance algorithm is adapted to use this automaton. The total number of trie and failure arcs is significantly less than the number of arcs in the DFA version of the algorithm.

Derrick G. Kourie, Bruce W. Watson, Loek Cleophas, Fritz Venter: Failure Deterministic Finite Automata, pp. 28–41.
Proceedings of PSC 2012, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05095-8 © Czech Technical University in Prague, Czech Republic

Benchmarks reported in [22] suggest that the gain in space efficiency comes at the cost of about 20% reduction in processing speed.

In addition to their use in the Knuth-Morris-Pratt and Aho-Corasick keyword pattern matching algorithms, there has been some work in broadening the use of failure functions, including:

- Kowaltowski, Lucchesi and Stolfi [16] present failure functions (and algorithms for computing them) in the restricted case of *acyclic automata* – especially as used in various natural language processing applications, such as spell-checking.
- Mohri [18] presents algorithms for the restricted case of constructing a failure function and manipulating $\mathcal{D}$ such that the resulting FDFA accepts language $\Sigma^* \cdot \mathcal{L}(\mathcal{D})$. The resulting compact representation is primarily useful in pattern matching for the language $\mathcal{D}$ *somewhere* in an input string x, as the Σ^* matches the prefix of x before the match.
- Crochemore and Hancart [4] illustrate how failure arc placement can sometimes be further optimised, but they do not give a general construction algorithm for deriving an FDFA from a DFA.

Our work takes up the failure arc idea and generalises it. Below we describe an ordered approach to deriving a language-equivalent FDFA from *any complete* DFA. This generalisation brings several issues to the fore.

The starting point to address these issues is the provision of a formal definition of an FDFA and its associated language. In terms of this definition, a DFA can be viewed as a degenerate FDFA. The right language of an FDFA state is recursively defined, and this provides a formal definition of an FDFA's language. Starting with the DFA (seen as a degenerate FDFA), we then show how to build incrementally a sequence of language-equivalent FDFAs. At each increment a set of arcs is replaced with a single failure arc while preserving the right languages of all states involved in such a transformation. As a consequence, the language recognised by FDFAs produced from transformation to the next remains invariant.

The matter of which set of arcs to select for transformation at each next iteration step is non-trivial. In general, there will be many possibilities, different selections leading to different FDFAs. One of the complications is that so-called divergent cycles of failure arcs have to be avoided (although non-divergent cycles may be tolerated).

To ensure that all candidate arcs for transformation are identified, we turn to formal concept analysis (FCA) – a domain of study in which a so-called (formal) concept lattice identifies clusters of objects that share common attributes [2]. We show how information about a complete DFA can be encapsulated in what we call a state/out-transition concept lattice. The state/out-transition lattice isolates arc sets that could potentially be replaced by failure arcs, and the arc redundancy measure is used to prioritise which sets to first select for such replacement. In this sense, we approximate a greedy algorithm. We also indicate how to proceed in order to render the algorithm a strictly greedy one, at some cost to the algorithm's efficiency. Since the greediness does not guarantee optimality, finding an efficient algorithm for deriving an arc-minimal language-preserving FDFA remains an open problem.

In summary, then, Section 2 provides the necessary formal material about FDFAs, while Section 3 introduces the reader to the FCA theory about concept lattices that is needed in this paper. Section 4 then shows how to build a state/out-transition concept lattice from a DFA. It also provides an algorithm which uses such a lattice to derive a language equivalent FDFA. Because the resulting FDFA retains all the

original DFA states, the algorithm is characterised as a DFA-homomorphic algorithm. In Section 5 we point to additional work on this theme that is currently on our agenda. This includes algorithms under development which introduce failure arcs to new states derived directly from the lattice. Because of this, these may be described as lattice-homomorphic algorithms. However, full elaboration of these algorithms will be provided in subsequent research contributions.

2 Failure Deterministic Finite Automata

In defining and discussing an FDFA below, we rely on the following conventions and general notation.

- Where convenient, a function will be regarded as a set of pairs, the first element being from its domain and the second from its range. A function which is not guaranteed to be total but may be, is called a *possibly partial* function.
- The domain of any function f is denoted by $\operatorname{dom} f$. If $q \notin \operatorname{dom} f$ for the possibly partial function f, then this is denoted by $f(q) = \perp$.
- A DFA denoted by $\mathcal{D} = (Q, \Sigma, \delta, F, s)$ is considered, where Q is the DFA's set of states; Σ is its alphabet; δ is the transition function mapping state / symbol pairs to a new state; s is the start state; and F is the set of final states.
- We use $\Sigma_q = \{a : \delta(q, a) \neq \perp\}$ to denote symbols labeling out-transitions of state q, and $\not\Sigma_q$ for $\Sigma \setminus \Sigma_q$. A complete DFA (sometimes called a total DFA, because δ is a total function) is characterised by the fact that $\Sigma_q = \Sigma$ for all $q \in Q$. Note that any DFA can easily be converted to a language-equivalent complete DFA by simply introducing an arc to a common sink state for every state q and symbol a such that $\delta(q, a) = \perp$.
- We will also use the function $\mathit{head} : \Sigma^+ \to \Sigma$ where $\mathit{head}(av) = a$; and the function $\mathit{tail} : \Sigma^+ \to \Sigma^*$ where $\mathit{tail}(av) = v$.
- We define the extended transition function $\delta^* : Q \times \Sigma^* \to Q$ by $\delta^*(q, w) = q$ if $w = \varepsilon$ and by $\delta^*(q, w) = \delta^*(\delta(q, \mathit{head}(w)), \mathit{tail}(w))$ otherwise.
- Given δ^*, the language of $\mathcal{D}$ is defined by $\mathcal{L}(\mathcal{D}) = \{w \mid \delta^*(s, w) \in F\}$.
- If $L \subseteq \Sigma^*$ and $u \in \Sigma$ then $u \cdot L$ denotes the prefixing of all elements in L by the symbol u, i.e. $u \cdot L = \{uw : w \in L\}$. Of course, $u \cdot \emptyset = \emptyset$.

Definition 1 (FDFA). *$\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ is an FDFA if $\mathfrak{f} : Q \to Q$ is a possibly partial function and $\mathcal{D} = (Q, \Sigma, \delta, F, s)$ is a DFA.*

We shall call $\mathcal{D}$ the *embedded DFA* of $\mathcal{F}$ and $\mathfrak{f}$ the *failure function* of $\mathcal{F}$. If $q \in \operatorname{dom} \mathfrak{f}$, then q is called a *failure state*.

Definition 2 (Right language of an FDFA's state). *The right language of state q in FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$, denoted by $\overrightarrow{\mathcal{L}}(\mathcal{F}, q)$, is defined as the smallest language such that $\overrightarrow{\mathcal{L}}(\mathcal{F}, q) = \overrightarrow{\mathcal{L}}_\delta(\mathcal{F}, q) \cup \overrightarrow{\mathcal{L}}_\mathfrak{f}(\mathcal{F}, q)$, where*

$$\overrightarrow{\mathcal{L}}_\delta(\mathcal{F}, q) = \left(\bigcup_{b \in \Sigma_q} b \cdot \overrightarrow{\mathcal{L}}(\mathcal{F}, \delta(q, b)) \right) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

$$\overrightarrow{\mathcal{L}}_\mathfrak{f}(\mathcal{F}, q) = \begin{cases} \overrightarrow{\mathcal{L}}(\mathcal{F}, \mathfrak{f}(q)) \cap (\not\Sigma_q \Sigma^*) & \text{if } \mathfrak{f}(q) \neq \perp \\ \emptyset & \text{otherwise} \end{cases}$$

Thus, the right language of an FDFA in state q, written $\overrightarrow{\mathcal{L}}(\mathcal{F}, q)$, consists of three components: (1) all strings that can be generated from that state by making a conventional DFA transition to the next state on one of the out-transition symbols in Σ_q; (2) ε if q is final; and (3) those words in $\overrightarrow{\mathcal{L}}(\mathcal{F}, \mathfrak{f}(q))$ (the right language of the next state as determined by the failure function at q) that begin with a symbol *not* in Σ_q, because any word beginning with a symbol in Σ_q would already have caused a conventional DFA transition from q.

(Such a recursive definition of right language is well-formed. The above definition essentially gives rise to a finite set of equations with variables $\overrightarrow{\mathcal{L}}(\mathcal{F}, q)$, $\overrightarrow{\mathcal{L}}_\delta(\mathcal{F}, q)$ or $\overrightarrow{\mathcal{L}}_\mathfrak{f}(\mathcal{F}, q)$ (for all states q) on the left-hand side. All of those equations are either right-linear or chain-rules, and Gaussian elimination/substitution can be used to partially solve them, leaving a limited number of self- or mutually-recursive equations. Those equations are solvable (as a regular language) using Arden's lemma [20, Lemma 2.9, page 100]. See [20, Section 4.3.1, page 133] for a detailed example resembling this one.)

Definition 3 (Language of an FDFA). *The language of an FDFA $\mathcal{F}$ is denoted by $\mathcal{L}(\mathcal{F})$ and is defined as $\overrightarrow{\mathcal{L}}(\mathcal{F}, s)$, where s denotes the start state of $\mathcal{F}$.*

Definition 4 (FDFA equivalence). *An FDFA $\mathcal{D}$ (which may possibly be a DFA) is said to be* (language) equivalent *to an FDFA $\mathcal{F}$ iff $\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{D})$. This will be denoted by $\mathcal{F} \equiv \mathcal{D}$.*

Clearly, the embedded DFA of an FDFA is not, in general, equivalent to the FDFA, but it is in the degenerate case, i.e. when $\mathfrak{f} = \emptyset$. As previously noted, a DFA can therefore be seen as a special case of an FDFA – it is an FDFA that has a degenerate failure function.

Note that for a given FDFA, there could be many equivalent DFAs and vice versa. It is well known that the regular languages partition the set of DFAs into equivalence classes. Thus, each regular language $\mathcal{R}$ defines a class $\mathcal{E}_\mathcal{D}(\mathcal{R}) = \{\mathcal{D} \mid \mathcal{D} \text{ is a DFA} \wedge \mathcal{L}(\mathcal{D}) = \mathcal{R}\}$ that is disjoint from every other such class. Similarly, the regular languages also induce equivalence classes of FDFAs so that for regular language $\mathcal{R}$ there is a unique and partitioning set of FDFAs $\mathcal{E}_\mathcal{F}(\mathcal{R}) = \{\mathcal{F} \mid \mathcal{F} \text{ is an FDFA} \wedge \mathcal{L}(\mathcal{F}) = \mathcal{R}\}$. Since every DFA is a degenerate FDFA, $\mathcal{E}_\mathcal{D}(\mathcal{R}) \subseteq \mathcal{E}_\mathcal{F}(\mathcal{R})$.

The algorithm proposed in Section 4 may be thought of as starting off with $\mathcal{D} \in \mathcal{E}_\mathcal{D}(\mathcal{R})$ and deriving a sequence of $\mathcal{F}_i \in \mathcal{E}_\mathcal{F}(\mathcal{R})$, terminating when there are no further opportunities for removing elements of δ while adding elements of $\mathfrak{f}$.

The FDFA $\mathcal{F}$ produced by that algorithm can be used for recognising whether a string x is a member of $\mathcal{L}(\mathcal{F})$. Algorithm 1 shows how this can be done. It assumes FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ is given.

In this text, the Guarded Command Language (GCL) is used to specify algorithms. This minimalist and easy to use specification language was invented by Dijkstra [7] and remains widely in use because of its conciseness and precision [14].

We rely on GCL's multiple guarded command format for the loop. In this form, the loop comprises of two guarded commands of the form $G \rightarrow S$ where G is a boolean expression and S is a command. All guards are evaluated at each iteration, and a statement is non-deterministically selected among those whose guards evaluate to **true**. If no guard evaluates to **true**, the loop terminates[1].

[1] **cand** and **cor** stand for "conditional and" and "conditional or" respectively, i.e. the equivalent of the short circuit operators && and || in C++, Java, etc.

The virtue of this multiple guarded command loop format is that it highlights the symmetry with standard DFA acceptance. The standard algorithm is identical to Algorithm 1, but with the second loop guard absent.

Algorithm 1 (Test for string membership of an FDFA's language)
$y, q := x, s$;
{ ***Invariant:*** *y is untested and the current state is q* }
do $(y \neq \varepsilon)$ **cand** $(\delta(q, head(y)) \neq \bot) \;\rightarrow\; q, y := \delta(q, head(y)), tail(y)$
[] $(y \neq \varepsilon)$ **cand** $((\delta(q, head(y)) = \bot) \;\wedge\; (\mathfrak{f}(q) \neq \bot)) \;\rightarrow\; q := \mathfrak{f}(q)$
od;
{ $((y = \varepsilon)$ **cor** $((\delta(q, head(y)) = \bot) \wedge (\mathfrak{f}(q) = \bot)))$ }
$accept := (y = \varepsilon) \wedge (q \in F)$
{ **post** $(accept \Leftrightarrow x \in \mathcal{L}(\mathcal{F}))$ }

However, Algorithm 1 embodies a potential complication that does not arise in its DFA counterpart. The presence of cycles in the failure function could lead to complications. In order to understand the meaning and consequences of such cycles, we begin by defining the notion of a failure path.

Definition 5 (Failure path and failure alphabet). *A sequence of FDFA states, $\langle p_0, p_1, \ldots, p_n \rangle$ of length $n > 0$ is called a* failure path *from p_0 to p_n, written $p_0 \overset{\mathfrak{f}}{\rightsquigarrow} p_n$, iff $\forall i \in [0, n) : \mathfrak{f}(p_i) = p_{i+1}$. For such a failure path, $\Sigma_{p_0 \overset{\mathfrak{f}}{\rightsquigarrow} p_n} = \not\Sigma_{p_0} \cap \not\Sigma_{p_1} \cap \cdots \cap \not\Sigma_{p_{n-1}}$ is its* failure alphabet.

Where convenient, $p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_j$ will be used as a predicate to assert that the FDFA under consideration has a failure path from state p_i to state p_j.

In Algorithm 1, the transition which occurs on symbols in Σ_q is determined by δ, and if q is a failure state then the transition to occur on symbols in $\not\Sigma_q$ is determined by the failure function $\mathfrak{f}$. The failure alphabet of a failure path is therefore the set of symbols, each of which is guaranteed to cause failure transitions from the start of the failure path to its end. This insight becomes important in distinguishing between failure paths that form cycles. We shall simply call a failure path that forms a cycle a *failure cycle* and designate it by $p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i$, where p_i is any state in the cycle.

Definition 6 (Divergent failure cycle). *A failure cycle $p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i$ is* divergent *iff $\Sigma_{p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i} \neq \emptyset$.*

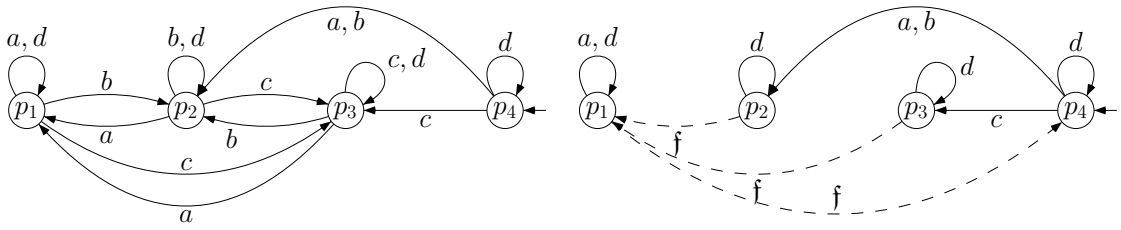
The term *divergent* is inspired by its use in process algebras. In that domain, a divergent concurrent system is one that is trapped into a cycle of non-productive state changes [19]. A divergent failure cycle in an FDFA would cause analogous behaviour in Algorithm 1. If the algorithm is examining symbol a in state p_i, where $a \in \Sigma_{p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i}$, then the algorithm will cycle non-productively through the divergent failure cycle without ever consuming symbol a. Clearly, therefore, it is advisable to avoid divergent failure cycles when constructing an FDFA. On the other hand, cycles which are not divergent (i.e. where $\Sigma_{p_i \overset{\mathfrak{f}}{\rightsquigarrow} p_i} = \emptyset$) are harmless, since it is guaranteed that at some state in the cycle, a symbol will *eventually* be consumed.

Definition 1 of an FDFA is as general as possible. It does not preclude failure cycles, whether or not they are divergent. It allows for useless states and transitions, including useless failure transitions. For example, a failure arc from state q where $\Sigma_q = \Sigma$ serves no purpose, but is not prohibited in the FDFA definition. We do not consider such cases in detail here, but ensure that they are avoided in the FDFA construction algorithm to be described.

Note in passing that the failure function description in [4] also allows for failure arcs in a general DFA setting, but no algorithm for constructing FDFAs in general is presented, and all failure arc cycles are prohibited there, even if they are not divergent. From what has been discussed above, this would seem to be an overly strict requirement.

Algorithm 1 operates in $\mathcal{O}(|w|)$ time in the best case, but in the worst case it has to traverse the path of an entire failure cycle before having a symbol of w consumed. Since the longest possible non-divergent cycle is $|Q|-1$, the algorithm's worst case performance is described by $\mathcal{O}(|w| \times (|Q| - 1))$. The corresponding DFA string membership algorithm operates in $\mathcal{O}(|w|)$.

However, there is a potential savings in arc storage if an FDFA is used instead of a DFA. For example, consider the DFA depicted in Figure 1a. It has a total of sixteen arcs. (Doubly labelled arcs are counted twice, because storage is required to represent each transition.) The FDFA in Figure 1b is language equivalent to the DFA



(a) Initial DFA: $|\delta| = 16, |\mathfrak{f}| = 0$

(b) FDFA after two iterations of Algorithm 2: $|\delta| = 8, |\mathfrak{f}| = 3$

Figure 1: Initial DFA and an equivalent FDFA. All states are considered final

in Figure 1a. The FDFA has only eight arcs, and three failure transitions (represented by dashed arcs). This saving in arcs is possible because a conventional DFA sometimes contains redundancies, i.e. it may have transitions to the same state from several destinations, all on the same symbol[2]. For example, in Figure 1a, all states make a transition to state p_1 on a, the transition from p_4 on a being an exception. All states transition to state p_2 on b, and all states transition to state p_3 on c.

The FDFA in the Figure 1b is designed to handle transitions that are unique at each state, and to *fail over* to another state if the transition to be made on a set of symbols is shared with other states. For example, in state p_2, a transition on d is determined locally, yet on all other symbols, a failure transition is made to p_1, since on those symbols the behaviour from the states is the same.

[2] DFA *minimization* relies on such redundancy, but only works in case of right language *equality* between states, vs. *containment* in the FDFA case.

Thus, to recognise the string *abca*, the following DFA transitions are made in Figure 1a

$$p_4 \xrightarrow{a} p_2 \xrightarrow{b} p_2 \xrightarrow{c} p_3 \xrightarrow{a} p_1$$

However, in the case of the FDFA in Figure 1b the transitions made are as follows

$$p_4 \xrightarrow{a} p_2 \xrightarrow{\mathfrak{f}} p_1 \xrightarrow{\mathfrak{f}} p_4 \xrightarrow{b} p_2 \xrightarrow{\mathfrak{f}} p_1 \xrightarrow{\mathfrak{f}} p_4 \xrightarrow{c} p_3 \xrightarrow{\mathfrak{f}} p_1 \xrightarrow{a} p_1$$

An FDFA therefore needs at most $\mathcal{O}(|Q| \times (\Sigma + 1))$ to store δ and $\mathfrak{f}$. However, the actual storage will be decreased from this worst case estimate to the extent that δ can be minimized when constructing the FDFA. The challenge taken up here, therefore, is to derive from a DFA (seen here as a degenerate FDFA) say $\mathcal{F}' = (Q', \Sigma, \delta', \emptyset, s', F')$, an equivalent FDFA, say $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, s, F)$ such that $|\delta'| - (|\delta| + |\mathfrak{f}|)$ is as large as possible. Because of the benchmarking results reported in [22], we conjecture that the time penalty will be about 20%.

For the purposes of the algorithm, we assume $\mathcal{F}'$ to be a complete DFA, i.e. for every state q, $\Sigma_q = \Sigma$. Furthermore, we regard the various states as constants (i.e. $Q = Q', s = s'$ and $F = F'$). The algorithm thus preserves the originating DFA's shape, and will, for this reason, be called a DFA-homomorphic algorithm. In the algorithm δ and $\mathfrak{f}$ are variables whose values change from their initial values δ' and $\mathfrak{f}'$. The algorithm also ensures that at every step the right language of every state remains unchanged.

A theorem which relies on a predicate FailPred(P, q, X) indicates conditions under which the right language is preserved. The predicate is defined as follows.

Definition 7 (FailPred(P, q, X)). *For $P \cup \{q\} \subseteq Q$ and $X \subseteq \Sigma$, FailPred(P, q, X) is defined by*

$$\begin{aligned}
\forall p \in P : (\\
&(\Sigma_p = \Sigma) && (1)\\
\wedge\ &(\mathfrak{f}(p) = \perp) && (2)\\
\wedge\ &(\forall (a \in X) : (\delta(p, a) = \delta(q, a))) && (3)\\
\wedge\ &(q \overset{\mathfrak{f}}{\rightsquigarrow} p \Rightarrow (\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap X = \emptyset)) && (4)\\
)
\end{aligned}$$

A scenario in which this predicate holds is sketched in Figure 2a, where it is assumed that $P = \{p\}$, $\Sigma = \{a, b, c\}$ and $X = \{a, b\}$. Notice that the scenario in the figure complies with the first three conjuncts of Definition 7, i.e. $(\Sigma_p = \Sigma)$, complying with conjunct 1; $(\forall a \in X : (\delta(p, a) = \delta(q, a))$, complying with conjunct 3 ; and $(\mathfrak{f}(p) = \perp)$, complying with conjunct 2. Furthermore, the figure shows that $\mathfrak{f}(q) = p$, and thus $q \overset{\mathfrak{f}}{\rightsquigarrow} p$. Clearly, $\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} = \not\Sigma_q = \{c\}$ and since $\{c\} \cap X = \emptyset$, conjunct 4 holds as well. Thus, Figure 2a depicts a scenario in which FailPred(P, q, X) holds.

Figure 2b shows the result of removing the a and b transitions from p, and providing a failure transition from p to q. Note that this can be done without disturbing the right languages of any of the states in the figures. Note also that because conjunct 4 holds, we can be sure that a divergent cycle has not been created. Theorem 8 generalises these observations.

(a) Before (b) After

Figure 2: Theorem 8 applied, where $X = \{a, b\}$

Theorem 8 (A transformation that preserves right languages and does not introduce any failure cycle). *Let $\mathcal{F}$ be an FDFA such that $P \cup \{q\} \subseteq Q$, $X \subseteq \Sigma$ and FailPred(P, q, X) holds. Then the following transformation on each $p \in P$ leaves the right languages of all states unchanged and does not introduce any failure cycle:*

> *Delete from δ all transitions from p on each symbol in X, and add a failure arc from p to q.*

Applying such a transformation to FDFA $\mathcal{F}'$ results in an FDFA $\mathcal{F}$ such that $\mathcal{F}' \equiv \mathcal{F}$, in which $|\delta|$ has decreased by $|X|$ and $|\mathfrak{f}|$ has increased by 1. Theorem 8 may be applied repeatedly, as long as P, q and X satisfying the definition above can be found. In Section 4 we will show how formal concept lattices, introduced in the next section, can be used to identify such P, q and X.

3 State / Out-Transition Formal Concept Lattices

A *formal concept lattice* can be defined in a domain of discourse consisting of a set of objects, and a set of attributes that the various objects possess. In such a domain, a *concept* is considered to be a pair of two sets: a set of objects, the concept's *extent*; and a set of attributes, the concept's *intent*. All objects in the concept's extent have in common all and only the attributes in the intent. Furthermore, the extent is maximal over the objects: there may not be any object outside of the concept's extent which also possesses all the attributes in the intent.

In the theory known as formal concept analysis, such concepts are considered to be partially ordered: if c_i and c_j are two arbitrary concepts in the domain, and if $ext(c)$ denotes concept c's extent, then $c_i \leq c_j \Leftrightarrow ext(c_i) \subseteq ext(c_j)$. Equality holds if and only if $i = j$. Furthermore, it can be shown that there is a duality in the role of objects and attributes, such that if $int(c)$ denotes concept c's intent, then $c_i \leq c_j \Leftrightarrow int(c_j) \subseteq int(c_i)$. The relationship between objects and attributes in a given domain can be presented as a cross table known as a *context*. An example is shown in Table 1. The rows represent the objects $p_1, \ldots, p_4$ and the columns represent attributes designated $\langle a, p_1\rangle$, $\langle a, p_2\rangle$, $\langle b, p_2\rangle$, $\ldots$, $\langle d, p_4\rangle$. (We discuss the reason for these rather strange attributes later.) An entry in a cell indicates that the relevant object has the indicated attributes. E.g. object p_4 has attributes $\{\langle a, p_2\rangle, \langle b, p_2\rangle, \langle c, p_3\rangle, \langle d, p_4\rangle\}$.

It can be shown that the partial ordering over all possible concepts implied by such a context, constitutes a lattice. Various lattice construction algorithms have been devised to extract all possible concepts from a given context and to arrange them in a graph structure that reflects their parent/child relationships [17,12]. Figure 3

shows a line diagram, generated from the context in Table 1, showing the ordering of concepts in the lattice. Concepts have been labelled $c1, c2, c3, c4, c123$ and $c1234$.

	$\langle a, p_1\rangle$	$\langle a, p_2\rangle$	$\langle b, p_2\rangle$	$\langle c, p_3\rangle$	$\langle d, p_1\rangle$	$\langle d, p_2\rangle$	$\langle d, p_3\rangle$	$\langle d, p_4\rangle$
p_1	1		1	1	1			
p_2	1		1	1		1		
p_3	1		1	1			1	
p_4		1	1	1				1

Table 1: The state/out-transition context of DFA in Figure 1a

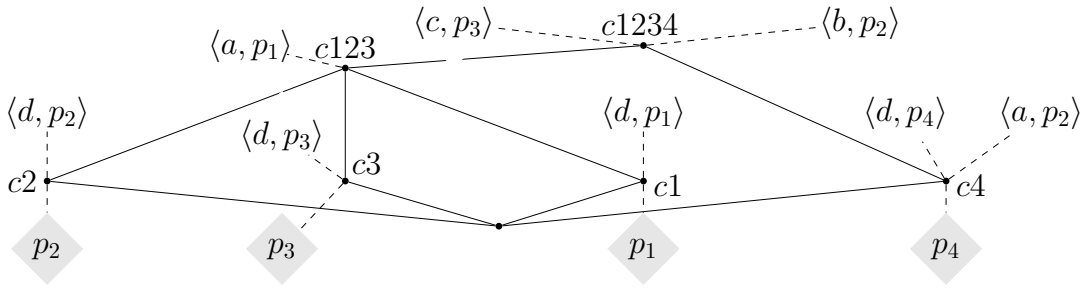


Figure 3: State/out-transition formal concept lattice of DFA in Figure 1a

Consider concept $c123$. Its extent is given by $ext(c123) = \{p_1, p_2, p_3\}$, while its intent is $int(c123) = \{\langle a, p_1\rangle, \langle c, p_3\rangle, \langle b, p_2\rangle\}$. Thus, concept $c123$ indicates that objects p_1, p_2 and p_3 share all and only the attributes $\langle a, p_1\rangle$, $\langle c, p_3\rangle$ and $\langle b, p_2\rangle$.

Concept $c123$ illustrates that the extent of a concept is the union of the extents of its children, together with any of its so-called "own objects". In this case, $c123$ does not have any own objects. Its children are $c1, c2$ and $c3$, and their respective extents correspond to their own objects, which are explicitly shown in the diagram – i.e. their extents are $\{p_1\}, \{p_2\}$ and $\{p_3\}$ respectively. Dually, concept $c123$ also illustrates the fact that the intent of a concept is the union of the intents of all its parents, together with any of its so-called "own attributes". It has $\langle a, p_1\rangle$ as its single own attribute, and its only parent, $c1234$, adds its intent, $\{\langle c, p_3\rangle, \langle b, p_2\rangle\}$, to that of $c123$.

Information in a DFA's transition graph can be represented in a context, and hence in a formal concept lattice. Here we propose one particular way to do so and call the result a *state/out-transition (formal concept) lattice*. For a DFA D, we denote this lattice by $\mathcal{SO}(\mathcal{D})$. The set of objects in $\mathcal{SO}(\mathcal{D})$ is simply the set of states in D, namely Q. Each attribute is a pair consisting of the label of an out-transition from some state, and the corresponding destination state. Formally, $\langle b, p\rangle$ is an attribute in $\mathcal{SO}(\mathcal{D})$ iff $\exists : q \in Q : \delta(q, b) = p$. In this case, $\langle b, p\rangle$ is an attribute of object q. The context in Table 1 was derived from the DFA in Figure 1a in precisely this way, and hence Figure 3 shows $\mathcal{SO}(\mathcal{D})$.

The space and time requirements for building the lattice's context table are determined by the size of δ, i.e. they are $\mathcal{O}(|Q|^2 \times |\Sigma|)$. An SO-lattice is a constrained lattice in the sense that its objects are constrained to each have exactly one attribute from each of $|\Sigma|$ classes, each class having $|Q|$ attributes. In [13] it is shown that the number of concepts for such a lattice is bound from above by $min((1 + |\Sigma|)^{|Q|}, \frac{|Q|}{(1+|\Sigma|)} 2^{1+|\Sigma|})$.

For convenience, we shall denote this expression by $\mathcal{LB}(\Sigma, Q)$. This means that for a fixed alphabet, an upper bound of the lattice size eventually becomes linearly dependent on the number of states.

4 A DFA-Homomorphic Algorithm

Consider an arbitrary concept c in $\mathcal{SO}(\mathcal{D})$, the state/out-transition lattice for a complete DFA $\mathcal{D}$. By definition, all states in $ext(c)$ share all and only the out-transitions in $int(c)$. (Of course, each state in $ext(c)$ may have other out-transitions.) For convenience, let $m = |ext(c)|$ and $n = |int(c)|$. Let q be any state in $ext(c)$, let $P = ext(c) \setminus \{q\}$ and let $X = \mathrm{dom}\, int(c)$. Thus $X \subseteq \Sigma$ is the set of symbols on which transitions to common states are made from all states in $ext(c)$.

We argue that FailPred(P, q, X) is true because the following holds for each $p \in P$. Conjunct 1 of the predicate is true because the DFA is assumed to be complete. Conjuncts 2 and 4 of the predicate hold because a DFA has no failure arcs. Conjunct 3 holds by the construction of $\mathcal{SO}(\mathcal{D})$ and by the definition of a concept in a concept lattice. Therefore Theorem 8 may be applied to produce an equivalent FDFA. This entails the following arc changes:

> For each $p \in P$ remove all outgoing arcs represented in $int(c)$. Thus, the number of arcs removed from $\mathcal{D}$ is $n(m-1)$.
> For each $p \in P$ install an outgoing failure arc to q. Thus, the number of arcs added to $\mathcal{D}$ is $(m-1)$.

As a result of these steps, $(n-1)(m-1)$ arcs will be removed from the initial structure.

For a given concept, c, we will call $(n-1)(m-1)$ its *arc redundancy*, denoted by $ar(c)$. For example, $ar(c123) = (3-1)\times(3-1) = 4$ since $|int(c123)| = |ext(c123)| = 3$.

If the above steps to construct a failure arc are applied to a concept c for which $ar(c) = 0$, there will be no decline in the number of arcs. Conversely, the 'maximum' decline is obtained if one selects from all the concepts, the one for which $ar(c)$ is 'maximal'. (Note that 'maximal' is used here in terms of the initially computed $ar(c)$ – it may not necessarily be maximal in terms of the current arc redundancy values, as we will discuss below.) This suggests the following 'greedy' algorithm for constructing FDFA $\mathcal{F}$ from DFA $\mathcal{D}$, assuming that $\mathcal{SO}(\mathcal{D})$ is available.

Algorithm 2
$\mathfrak{f}, O := \emptyset, Q$;
{ *Assume that AR is set of concepts with non-zero arc redundancy* }
{ *Invariant:* $(\mathrm{dom}\,\mathfrak{f} = Q \setminus O)$ $\wedge$ (*Concepts in AR have not been processed*) }
do $((O \neq \emptyset) \wedge (AR \neq \emptyset)) \rightarrow$
  $c := maxcar(AR)$;
  $AR := AR \setminus \{c\}$;
  let $q \in ext(c)$;
  $P := ext(c) \setminus \{q\}$;
  for each $(p \in P \cap O) \rightarrow$
    if $\neg(q \overset{\mathfrak{f}}{\rightsquigarrow} p)$ ***COR*** $((\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap \mathrm{dom}\, int(c)) = \emptyset) \rightarrow$
      for each $(\langle a, r\rangle \in int(c)) \rightarrow$
        $\delta := \delta \setminus \{\langle p, a, r\rangle\}$
      rof;

$\mathfrak{f}(p), O := q, O \setminus \{p\};$
$[\!]$ $(q \overset{\mathfrak{f}}{\rightsquigarrow} p)$ ***CAND*** $((\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap \operatorname{dom} int(c)) \neq \emptyset) \rightarrow$ **skip**
fi
rof
od

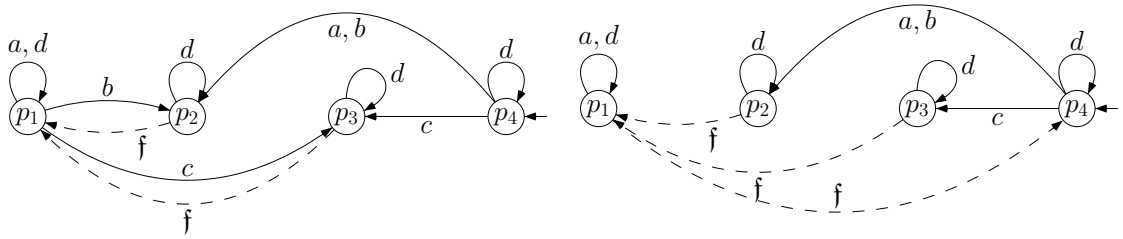
The set, AR, of concepts with non-zero arc redundancy is easily computed, and is assumed to be available to the algorithm. The algorithm initialises and maintains the set O of states which do not originate failure transitions, i.e. O is defined by $\operatorname{dom} \mathfrak{f} = Q \setminus O$. A function $maxcar : \mathcal{P}(\mathcal{SO}(\mathcal{D})) \rightarrow \mathcal{SO}(\mathcal{D})$ is assumed which selects from AR a concept, c with the maximum arc redundancy as initially determined.

Note that, as stated above, q is arbitrarily selected from $ext(c)$ to act as the target for failure arcs. The outer **for each** loop identifies states remaining in $ext(c)$ which may serve as sources of failure arcs. Such states have to be in O (to ensure compliance with conjunct 2 of Definition 7). The **if** statement then ensures that δ arcs are replaced by $\mathfrak{f}$ (within the inner **for each** loop) if and only if conjunct 4 of Definition 7 holds[3], thus ensuring that divergent cycles are never produced.

Applying the algorithm to the DFA in Figure 1a, and making use of the state/out-transition lattice shown in Figure 3 yields the FDFA shown in Figure 4a after the first iteration of the outer **do** loop. To see that this is so, note that upon entering the loop, $AR = \{c123, c1234\}$ where $ar(c123) = 4$, $ar(c1234) = 3$ and $O = \{p_1, p_2, p_3, p_4\}$. Thus, in the first iteration $maxcar(AR)$ returns concept $c123$ and the algorithm removes $c123$ from AR. Choosing $q = p_1$ (any element of $P = \{p_1, p_2, p_3\}$ could have been chosen) as the destination of all failure nodes in this iteration, the **for each** loop removes the following 6 arcs (δ mappings) from the DFA in Figure 1a:

$$\{\langle p_2, a, p_1\rangle, \langle p_2, b, p_2\rangle, \langle p_2, c, p_3\rangle, \langle p_3, a, p_1\rangle, \langle p_3, b, p_2\rangle, \langle p_3, c, p_3\rangle\}$$

Thereafter, it inserts two failure transitions: $\{\langle p_2, p_1\rangle, \langle p_3, p_1\rangle\}$. It also changes O to $\{p1, p4\}$. As a result, the number of arcs has been reduced by 4 – as predicted by $c123$'s arc redundancy.



(a) After first iteration: $|\delta| = 9, |\mathfrak{f}| = 2$ (b) After second iteration: $|\delta| = 8, |\mathfrak{f}| = 3$

Figure 4: FDFA's as Algorithm 2 progresses

---

[3] The guard of the **if** statement, namely $\neg(q \overset{\mathfrak{f}}{\rightsquigarrow} p)$ **COR** $(\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap \operatorname{dom} int(c) = \emptyset))$, is logically equivalent to $(q \overset{\mathfrak{f}}{\rightsquigarrow} p) \Rightarrow (\Sigma_{q \overset{\mathfrak{f}}{\rightsquigarrow} p} \cap \operatorname{dom} int(c) = \emptyset)$, which in turn corresponds to conjunct 4 in Definition 7 in which $\operatorname{dom} int(c)$ takes the role of X.

After the second iteration of the outer **do** loop the FDFA in Figure 4b is obtained. (It is a copy of Figure 1b, reproduced here for convenience.) Upon entering the loop for a second time, $AR = \{c1234\}$. $maxcar(AR)$ therefore returns concept $c1234$. At this point one of the states in $ext(c1234) = \{p_1, p_2, p_3, p_4\}$ has to be selected as the destination of all failure nodes in this iteration.

The **for each** loop removes from the FDFA in Figure 4a the arcs $\langle p_1, b, p_2\rangle$ and $\langle p_1, c, p_3\rangle$ and inserts failure transition $\{\langle p_1, p_4\rangle\}$, reducing the number of arcs by 1.

We offer the following reflections, based on the algorithm and the example just given.

1. Out-transitions from p_2 and p_3 are not considered in the outer **for each** loop, since these became failure states in the previous iteration and were removed from set O. To have more than one failure arc emanating from a state would mean that we could no longer speak of a failure *function*, and we would not know under which circumstances which failure arc should be selected. This, of course, is the reason for conjunct 2 in Definition 7.
2. Suppose that instead of selecting p_4 from $ext(c1234)$ as the failure arc destination, p_2 had been chosen. In this case, p_1 and p_4 would be candidate source states for failure arcs to p_2. (Again, p_3 would be eliminated because it is already a failure state.) The **if** statement within the **for each** loop would discover that a failure arc $\langle p_1, p_2\rangle$ would result in a divergent failure cycle. Consequently, only failure transition $\langle p_4, p_2\rangle$ would be added, and arcs $\langle p_4, c, p_3\rangle$ and $\langle p_4, b, p_2\rangle$ would be removed.
3. It can easily be verified that the reduction in the number of arcs in the second iteration is by 1, no matter which member of $ext(c1234)$ is selected for the failure arc destination. This does *not* correspond to the initially computed value of $ar(c1234)$, namely 3. This is to be expected, because the algorithm as given above computes concept arc redundancy only once – at the start of the algorithm. Consequently, the algorithm in its above format naïvely ignores the fact that whenever states are removed from O, the concept arc redundancies may change for those concepts whose extents contain removed states. By implication, therefore, *maxcar* is no longer guaranteed to choose as "greedily" as it might have. This potential selection inefficiency could easily be overcome at the cost of recomputing concept arc redundancy whenever s is removed from O. In such a case, the arc redundancy of a concept whose extent contains s should account for the fact that s is not a candidate for being the *source* of an second failure arc.
4. The way in which the *target* state for failure arcs, q, is selected in Algorithm 2 could also be optimised to enhance the algorithm's greediness. Instead of selecting an arbitrary state in $ext(c)$, preference should be given to failure states (i.e. states already in $\mathrm{dom}\,\mathfrak{f}$). The reason for this heuristic is clear: when a state becomes a source state (i.e. a failure state), its δ arcs are removed, but when a state becomes a target state, no δ arcs are removed. Since an existing failure state is no longer a candidate for becoming a source state, and therefore cannot contribute to the removal of δ arcs, it might as well serve as the target of newly installed failure arcs, thus allowing more states to become source states and thus allowing more states to shed some of their δ arcs. If this heuristic is to be applied, then the recomputation of arc redundancy mentioned in point 3 above should be suitably adjusted.
5. The test to be carried out in the **if** statement of Algorithm 2 entails determining whether a divergent cycle will arise if a failure arc is installed from state p to

state q. Since cycle-determination is a well known task in general data structure theory, details here are unnecessary. In the present context, a cycle cannot be longer than $|Q|$. Furthermore cycles in a given FDFA have to be disjoint, since $\mathfrak{f}$ is a function. This considerably simplifies the task of identifying divergent cycles.

6. Consider the implications of providing a sink node to render a partial DFA complete so that it may serve as input to Algorithm 2 to produce an FDFA. Of course, there can be no guarantee that the number of arcs in the FDFA will be less than in the originating partial DFA. However, it may be possible to remove some of the inserted arcs from the FDFA. In particular, all arcs from non-failure states to the sink state may safely be removed. This applies, even if the non-failure state serves the target state of one or more failure states. Furthermore, noting that the sink state will have loops back to itself on all symbols (as part of the completeness requirement), it is possible that the sink state becomes the target of failure arcs from other states. Such failure arcs could be removed as well. Notwithstanding these few brief observations, the matter of converting partial DFAs to FDFAs requires further study.

Additional details relating to algorithmic enhancements mentioned in points 3 and 4 above are briefly taken up in [15], and an example is also given to illustrate the points that are made.

Recall from the previous section that the number of concepts in lattice $\mathfrak{SO}(\mathcal{D})$ is bound from above by $\mathcal{LB}(\Sigma, Q)$, giving a very rough upper bound for $|AR|$ in our algorithm. We expect the actual bound to be much lower than this, since it is not clear that a state/out-transition lattice can reach the upper bound mentioned, and many concepts may have no arc redundancy and thus not end up in AR. Nevertheless, using that bound, and noting that $O \subseteq Q$, the outer **do** loop is executed at most $\mathcal{LB}(\Sigma, Q)$ times under the assumption that AR is never recomputed and under the unrealistic assumption that all concepts initially have a positive arc redundancy. This bound also ignores the fact that the loop terminates when O becomes the empty set.

The outer **for each** loop is executed at most $|Q|$ times, as both P and O are subsets of Q. The complexity of computing the value of the guards of the **if** statement is bounded by the maximum of $|Q|$ (for failure path tracing) and $|\Sigma|$ (for checking intersection), while the inner **for each** loop has complexity at most $|\Sigma|$. Combining this gives $\mathcal{LB}(\Sigma, Q) \times |Q| \times max(|Q|, |\Sigma|) \times |\Sigma|)$ as a very coarse upper bound on the algorithm's time complexity.

5 The Next Steps

The foregoing has stimulated a number of future research questions and ideas relating to FDFAs, their properties, their relation to DFAs, and their construction. They include investigating transition and state minimality properties of FDFAs compared to their DFA counterparts; directly constructing an FDFA from a regular expression; handling partial DFAs; and constructing a DFA from a given FDFA. Additionally, we are investigating alternative construction algorithms for producing an FDFA that is language-equivalent to a given DFA. These also rely on a state/out-transition lattice, but allow for the generation of new states that are derived from lattice concepts. In this sense, they can be regarded as "lattice-homomorphic". Refinements of the DFA-homomorphic algorithm of Section 4 have also been developed. They relate to the recomputation of arc redundancy and to optimised selection strategies for target

state of a failure transition. The tradeoff between FDFA storage size reduction versus processing speed is currently under empirical investigation. We refer the reader to [15] for details regarding this and other future work.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18(6) June 1975, pp. 333–340.
2. C. Carpineto and G. Romano: *Concept Data Analysis: Theory and Applications*, John Wiley & Sons, England, 2004.
3. W. Coetser, D. G. Kourie, and B. W. Watson: *On regular expression hashing to reduce FA size.* IJFCS, 20(6) 2009, pp. 1069–1086.
4. M. Crochemore and C. Hancart: *Automata for matching patterns*, Springer-Verlag New York, Inc., New York, NY, USA, 1997, pp. 399–462.
5. J. Daciuk and D. Weiss: *Smaller representation of finite state automata*, in Proceedings of the Conference on Implementation and Application of Automata (CIAA), 2011, pp. 118–129.
6. N. de Beijer, L. Cleophas, D. G. Kourie, and B. W. Watson: *Improving automata efficiency by stretching and jamming*, in Proceedings of the Prague Stringology Conference (PSC), 2010, pp. 9–24.
7. E. W. Dijkstra: *Guarded commands, nondeterminacy and formal derivation of programs.* Commun. ACM, 18(8) Aug. 1975, pp. 453–457.
8. K. Driesen and U. Hölzle: *Minimizing row displacement dispatch tables*, in Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '95, New York, NY, USA, 1995, ACM, pp. 141–155.
9. E. Fredkin: *Trie memory.* Communications of the ACM, 3(9) 1960, pp. 490–499.
10. E. Ketcha Ngassam: *Towards cache optimization in finite automata implementations*, PhD thesis, University of Pretoria, 2007.
11. D. E. Knuth, J. James H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
12. D. G. Kourie, S. A. Obiedkov, B. W. Watson, and D. van der Merwe: *An incremental algorithm to construct a lattice of set intersections.* Science of Computer Programming, 74(3) 2009, pp. 128–142.
13. D. G. Kourie and G. D. Oosthuizen: *Lattices in machine learning: Complexity issues.* Acta Informatica, 35 1998, pp. 269–292.
14. D. G. Kourie and B. W. Watson: *The Correctness-by-Construction Approach to Programming*, Springer Verlag, 2012.
15. D. G. Kourie, B. W. Watson, T. Strauss, F. Venter, and L. Cleophas: *Failure deterministic finite automata*, Tech. Rep. 2012.1.0, FASTAR Research Group, 2012.
16. T. Kowaltowski, C. L. Lucchesi, and J. Stolfi: *Minimization of binary automata*, in Proceedings of the First South American String Processing Workshop, 1993, pp. 105–116.
17. S. O. Kuznetsov and S. A. Obiedkov: *Comparing performance of algorithms for generating concept lattices.* Journal of Experimental & Theoretical Artificial Intelligence, 14(2-3) 2002, pp. 189–216.
18. M. Mohri: *String-matching with automata.* Nordic Journal of Computing, 4 1997, pp. 217–231.
19. A. W. Roscoe: *The Theory and Practice of Concurrency*, Prentice Hall, 1997.
20. J. Sakarovitch: *Elements of Automata Theory*, Cambridge University Press, 2009.
21. R. E. Tarjan and A. C.-C. Yao: *Storing a sparse table.* Communications of the ACM ACM, 22(11) November 1979, pp. 606–611.
22. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Eindhoven University of Technology, September 1995.