An Assessment of Selected Algorithms for Generating Failure Deterministic Finite Automata

by

Madoda Nxumalo

Submitted in partial fulfillment of the requirements for the degree
Master of Science (Computer Science)
in the Faculty of Engineering, Built Environment and Information Technology
University of Pretoria, Pretoria

November 2015

Publication data:

Madoda Nxumalo. An Assessment of Selected Algorithms for Generating Failure Deterministic Finite Automata. Master's dissertation, University of Pretoria, Department of Computer Science, Pretoria, South Africa, November 2015.

Electronic, hyperlinked versions of this dissertation are available online, as Adobe PDF files, at:

http://www.fastar.org/

http://upetd.up.ac.za/UPeTD.htm

An Assessment of Selected Algorithms for Generating Failure Deterministic Finite Automata

by

Madoda Nxumalo
E-mail: mnxumalo@cs.up.ac.za

Abstract

A Failure Deterministic Finite Automaton (FDFA) offers a deterministic and a compact representation of an automaton that is used by various algorithms to solve pattern matching problems efficiently. An abstract, concept lattice based algorithm called the DFA - Homomorphic Algorithm (DHA) was proposed to convert a deterministic finite automata (DFA) into an FDFA. The abstract DHA has several nondeterministic choices. The DHA is tuned into four decisive and specialized variants that may potentially remove the optimal possible number of symbol transitions from the DFA while adding failure transitions. The resulting specialized FDFA are: MaxIntent FDFA, MinExtent FDFA, MaxIntent-MaxExtent FDFA and MaxArcReduncdancy FDFA. Furthermore, two output based investigations are conducted whereby two specific types of DFA-to-FDFA algorithms are compared with DHA variants. Firstly, the well-known Aho-Corasick algorithm, and its DFA is converted into DHA FDFA variants. Empirical and comparative results show that when heuristics for DHA variants are suitably chosen, the minimality attained by the Aho-Corasick algorithm in its output FDFAs can be closely approximated by DHA FDFAs. Secondly, testing DHA FDFAs in the general case whereby random DFAs and language equivalent FDFAs are carefully constructed. The random DFAs are converted into DHA FDFA types and the random FDFAs are compared with DHA FDFAs. A published non concept lattice based algorithm producing an FDFA called D^2FA is also shown to perform well in all experiments. In the general context DHA performed

well though not as good as the D^2FA algorithm. As a by-product of general case FDFA tests, an algorithm for generating random FDFAs and a language equivalent DFAs was proposed.

Keywords: Failure deterministic finite automaton, Failure transitions, Aho-Corasick, Random FDFA algorithm.

Supervisors : Prof. D. G. Kourie
Dr. L.G.W.A. Cleophas
Department : Department of Computer Science
Degree : Master of Science

To the memory of my parents

Acknowledgements

I would like to thank the following institutions and people for the support provided during the course of this research:

- Professor Derrick G. Kourie for the exceptional guidance, the mentorship and the support;
- Dr Loek Cleophas for the supervision and the ever available help;
- Professor Bruce W. Watson for sharing some valuable ideas towards this research;
- The RuZA collaboration for the exposure to a large scale research and special thanks to Alexey Neznanov for providing us with FCART;
- My family for the never ending support;
- My colleagues for the regular random discussions about this research and friends for the unceasing encouragement;
- The National Research Foundation (NRF) for the financial contribution.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

1.1 Motivation

This dissertation is concerned with certain types of finite automata (FAs) to be defined in Chapter 2. In general, FAs play an important role in many applications involving character sequence processing[1] of some kind, whether on natural language, network traffic, or biological sequence data. As stated by Watson [1], such automata may be exceptionally large, e.g. millions of states, and as a result consume much memory. Because of escalating amounts of data to be processed and limited memory space (especially in embedded devices), it is desirable to reduce the size of FAs.

One possible way of reducting memory requirements in FAs might be to rely on non-deterministic FAs (NFAs) instead of deterministic FAs (DFAs). However, reduction in space requirements should not incur a severe processing speed penalty. Because backtracking (which is costly in terms of processing time) is generally required when processing NFAs, they are probably not good candidates for space saving strategies.

[1] In character sequence processing, FAs are typically used to test whether a given string is a member of the associated language. The FA could also be used to *generate* one or more strings in the language. These tasks of membership testing or string generation are referred to here as *language processing*.

Saving DFA representation space can be done either with or without changing the structure of the DFA. By structure we mean the number of states and the source and destination of arcs between states. Reducing DFA memory space requirements without changing the arcs and/or states depends both on the underlying abstract representation of the DFA's arcs as well as on the data structure used to implement this representation. DFA transitions' representations include: adjacency lists, transition matrices and transition lists. They are described in detail by Crochemore and Hancart [2]. Moreover, the data structure to be used is typically associated with the DFA representation selected. For example, a transition matrix is generally represented by a two dimensional array (which, in turn, is usually implemented as a vector of vectors in a C/C++ environment). But there are other possible ways to represent DFAs — for example, using linked lists, if the matrix is very sparse. A transition list, on the other hand, is often stored in a hashing table or linked list.

Of course, as the DFA size increases, even the best data structure for the DFA will be limited by available computer memory space. In such contexts, it may be desirable to reduce the DFA's spatial representation requirements by reducing its number of arcs and/or states. In effect, this means replacing the original DFA with a language-equivalent new one that has fewer states and/or arcs. Two techniques to reduce the number states and/or arcs in a DFA, DFA minimisation and failure functions, are briefly described below.

Minimisation aims at transforming a deterministic automaton into a language equivalent deterministic automaton containing the minimal number of states. Note that this could also result in the removal of a substantial number of arcs. In general, minimising a DFA involves merging equivalent (non-distinct) states and removing unreachable states, resulting in a simpler DFA[2]. There are several general DFA minimisation algorithms. These include algorithms by Revuz [3], Brzozowski [4] and Hopcroft [5]. Some algorithms

[2]Essentially, states are equivalent if they have the same so-called *right language*. The definition of a state's right language and the details of how to identify equivalent states is beyond the scope of the present text.

minimise specific kinds of DFAs. For example, [1, 6, 7, 8] described specialised algorithms for converting acyclic DFAs to minimal acyclic DFAs. Because of their acyclic nature, such DFAs do not contain paths that form cycles and it is somewhat easier to determine whether states are equivalent.

An alternative approach to reducing DFAs memory requirements involves the use of a failure function to reduce the number of arcs. Here, states that share a set of outgoing regular arcs are identified and one of the states in this set is singled out as a destination. The common outgoing arcs at all the other states in the set are deleted and substituted by a set of special arcs called *failure arcs* (See [2, 9, 10].). See Figure 1.1 and Figure 1.2 for an example. In Figure 1.1 states q_1 and q_2 each have two exiting arcs, each bearing the same symbols as labels and leading to the same states. In Figure 1.2 state q_1's so-called symbol arcs have been replaced by a single failure arc from q_1 to q_2.

A *failure function* for a DFA defines a set of failure arcs on that DFA. A DFA that contains a failure function is called a *failure deterministic finite automaton* (*FDFA*) or in short simply called *failure-DFA*.

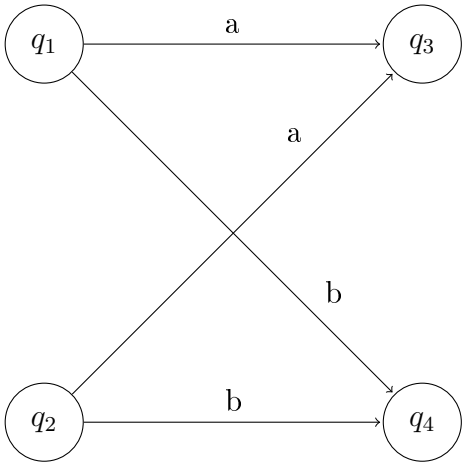

Figure 1.1: *DFA arcs*

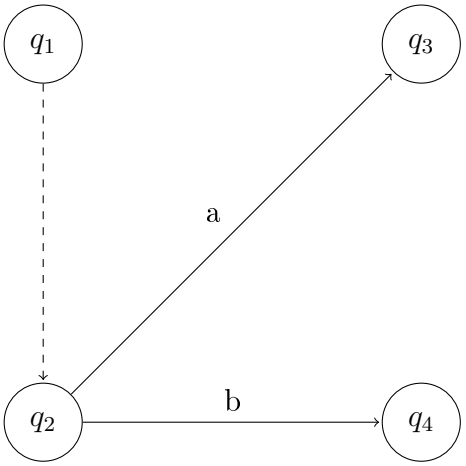

Figure 1.2: *FDFA arcs*

During language processing, if a state has no symbol arc for a given input symbol but is the source of a failure arc, then the failure arc is taken and the consumption of the input symbol at the current state is deferred. As

stated in [11, 12], such a symbol is therefore said to be "consumed" at some succeeding state. For example, consider Figure 1.2. Suppose that during language processing, a control was at state q_1 and a symbol a or b had to be processed, then the failure arc to state q_2 would be taken, and the symbol would be processed from that position, taking the control in the next step to either state q_3 for a or state q_4 for b.

Transforming a DFA into an FDFA by introducing failure arcs, results in the removal of some symbol arcs, possibly reducing the total number of arcs. For example, consider the two arcs from states q_1 and q_2 respectively that each have a transition on symbol a to another state q_3. (See Figure 1.1.) Since both these arcs are labelled by a and go to q_3, we can allow q_1 to have a default (failure) arc to q_2. This will only delay the execution of a, only to be processed at q_2. (Refer to Figure 1.2.)

Many arcs could be eliminated by using this approach, depending on the characteristics of arcs in the original DFA. Hence an FDFA may offer a compact space representation when compared to its source DFA. Moreover, an FDFA preserves the deterministic nature of the parent DFA—there is no ambiguity about what path to follow on a given symbol at any state. The research described in this dissertation is extensively concerned with this approach to reducing the size of a DFA.

In Kourie et al. [9], the term FDFA was introduced and it was pointed out that a DFA may be viewed as an FDFA that contains no failure arcs, that is a DFA may be viewed as a degenerate FDFA. Note, however, the use FDFAs — i.e. of failure arcs in a specialised DFAs — predates Kourie et al. [9]. These specialised FDFAs have been extensively applied to find solutions to a certain pattern matching problem, namely the problem of locating where a finite set of keywords occur in a given string. The classical Aho-Corasick (AC) automata by Aho and Corasick [13] which solve this so-called multiple keyword pattern matching problem may be viewed as specialised FDFAs. They are applied in network intrusion detection (refer to [14, 15]), in compiler construction (specifically in lexical analysis) (details in [16, 17]) and in text

mining (see [18]).

The Aho-Corasick algorithm executes a linear sweep through an input string to identify positions in the string where matches occur with elements of the given keyword set. There are two variants of the Aho-Corasick automata that may be used. The one variant is a DFA and the other is an FDFA. Details of how to construct the automata may be found in [19, Chapter 3.9] and [13]. It is known that the FDFA is minimal in the sense that no other language equivalent FDFA of the Aho-Corasick DFA can have fewer arcs. This known FDFA property, which predates Kourie et al. [9], was an important driver of the present research.

1.2 Problem Description

The basic objective of this research is to assess various concrete versions of the so-called *DFA-Homomorphic algorithm* (*DHA*) described in Kourie et al. [9]. The algorithm constructs from a complete[3] DFA a language-equivalent FDFA. The abstract version of the algorithm as reported in Kourie et al. [9] contains several nondeterministic choices. As a result, a concrete implementation of the algorithm requires that specific deterministic choices should be made at these points. Assessing the performance of various concrete versions entails determining how well each version does at replacing symbol arcs in the input DFA with failure arcs in the FDFA. The fewer the total number of arcs in the FDFA, the better that version of the algorithm is deemed to have performed.

The initial approach to assessing the performance of the DFA-homomorphic algorithm (DHA) relies on the previously mentioned minimality property of the Aho-Corasick FDFA. This minimal FDFA provides a convenient baseline for assessing the performance of concrete variants of the algorithm. That is, use an Aho-Corasick DFA as input to a variant of the DHA and compare the resulting FDFA to the minimal FDFA derived by the classical Aho-Corasick

[3] A complete DFA is such that *every* state has an arc on *every* symbol in the alphabet.

approach.

Subsequently, the performance of the algorithm and its variants (in terms of generating a minimal FDFA from a given DFA) will also be investigated in the general case. That is, in the case where the starting DFA has an *arbitrary* structure, instead of being limited to the specialised structure required in the Aho-Corasick case.

It should be noted that, at the start of this research, it had been assumed that DHA was the only general algorithm for generating language-equivalent FDFAs from DFAs. This appeared to be the perspective in the stringology, formal language theory and pattern matching research communities and literature. The initial research plan therefore focused on benchmarking this algorithm exclusively. However, well after many benchmark runs had been made, as part of my routine literature surveillance and exploration, I discovered references in the computer communications literature to algorithms that converted DFAs to so-called D^2FAs by Kumar et al. [12]. Closer examination showed that D^2FAs coincide precisely with FDFAs. It was therefore decided to include, as part of this research, some of the algorithms discussed in Kumar et al. [12] in the various benchmarking exercises.

1.3 Outline of the Dissertation

Chapter 2 provides the formal/mathematical preliminaries that will be used in this dissertation. These preliminaries include relevant definitions and notation for this work. Background literature is also discussed, with a keen focus on (failure) deterministic finite automata and formal concept analysis (FCA).

Chapter 3 presents the abstract DHA and its variants. A brief exploration of the DFA to FDFA transformation is presented. The transformation is explained based on how a DFA is represented as concepts of a concept lattice, which is then used to transform the DFA into an FDFA. Content under

discussion includes the modifications effected to the original algorithm to allow deterministic choices in the variant algorithms.

The research methods, theories, techniques and tools employed in the investigation of comparing FDFAs are described in Chapter 4. Problem domain restrictions are also discussed.

The experimental results related to comparing Aho-Corasick (AC) automata and DHA-FDFAs are presented in Chapter 5. The empirical results from all the variants of DHA are compared against the results of AC automata. Moreover, the FDFA results are discussed and critiqued.

An investigation of DHA-FDFAs built from general complete DFAs (as opposed to Aho-Corasick type DFAs) is undertaken in Chapter 6. Details are provided of how a language-equivalent complete DFA can be constructed from an FDFA that is "randomly" generated in a certain way. These DFAs are used as input to the DHA algorithm variants. The empirical results obtained examine the extent to which the FDFAs generated by DHA variants reconstruct the original "random" FDFA that was used to construct the DFA in the first place.

Finally, Chapter 7 gives the overall conclusions and suggests future research work ensuing from this dissertation.

Chapter 2

Preliminaries

This chapter defines the basic terminology and notations that will appear in the next chapters. Background research and related studies are also discussed here.

2.1 Definitions

In this section, definitions and notations are provided for concepts and terms relating to three domains of study: stringology, finite automata and formal concept analysis.

2.1.1 Stringology

As stated by Holub [20], *stringology* is the term that was first used by Galil in 1984 to describe the subfield of algorithmic research that is concerned with the processing of text strings. Notations commonly used in the stringology literature and adopted in this research is presented below.

Definition 2.1. *An* **alphabet** *is defined as a finite non-empty set of symbols. By convention, an alphabet is often denoted by* Σ*. The size of alphabet* Σ *is*

the number of symbols in Σ *and it is denoted by* $|\Sigma|$.

Definition 2.2. *A* **string** *(or word) over an alphabet is a finite sequence of symbols drawn from the alphabet. The length or size of a string* s *is denoted* $|s|$. *An empty string is denoted by* ϵ *and* $|\epsilon| = 0$.

Definition 2.3. Concatenation of strings: *Given two strings* p *and* q, *the concatenation of strings is represented as* $p.q$; *that is a new string is formed in which string* p *is precedes string* q.

Note that some literature sources use the form pq (without the dot notation) to denote concatenation of two strings p and q.

Definition 2.4. Types of substrings of a string: *If string* $s = p.q.v$ *then*

- q *is a* **substring** *or a* **factor** *of* s,
- $p.q$ *is a* **prefix** *of* s *and*
- $q.v$ *is a* **suffix** *of* s.
- *Moreover,* q *is a* **proper substring** *iff* $\neg((p = \epsilon) \wedge (v = \epsilon))$.
- *Similarly,* $p.q$ *is a* **proper prefix** *iff* $v \neq \epsilon$ *and*
- $q.v$ *is a* **proper suffix** *iff* $p \neq \epsilon$.

(Of course, p *and* v *are also prefixes and suffixes of* s *respectively.)*

Definition 2.5. *The* **Kleene closure** *of* Σ *is denoted by* Σ^* *and is defined as the set of all possible strings that can be formed from symbols in the alphabet* Σ, *including the empty string* ϵ. *Moreover, we define* $\Sigma^+ = \Sigma^* - \{\epsilon\}$

Definition 2.6. *The* **head** *operator (*$\texttt{head} \in \Sigma^+ \longrightarrow \Sigma$*) on a string is defined as:*

$$\texttt{head}(a.v) = a,$$

for $a \in \Sigma \wedge v \in \Sigma^*$.

Definition 2.7. *The* **tail** *operator (*$\texttt{tail} \in \Sigma^+ \longrightarrow \Sigma^*$*) on a string is defined as:*

$$\texttt{tail}(a.v) = v,$$

for $a \in \Sigma \wedge v \in \Sigma^*$.

Definition 2.8. *A* **language on alphabet** Σ *is defined as a finite subset of* Σ^*.

Definition 2.9. Concatenating two languages *is defined as follows; given two languages on an alphabet namely;* U *and* V *then*

$$U.V = \{u.v | u \in U \wedge v \in V\}$$

2.1.2 (Failure) Deterministic Finite Automata

Since this research is principally concerned with failure deterministic finite automata and their relationship to deterministic finite automata, the definitions below are limited to these kinds of automata. Specifically excluded, therefore, are definitions of non-deterministic finite automata or any other kind of finate state machine.

Definition 2.10. *A* **Deterministic Finite Automaton (DFA)** *is a quintuple denoted by* $\mathcal{D} = (Q, \Sigma, \delta, F, q_s)$*, where:*

- Q *is a finite set of states;*
- Σ *is an alphabet set (i.e. a finite set of symbols);*
- $\delta \in Q \times \Sigma \nrightarrow Q$ *is a (possibly partial) symbol transition function mapping state/symbol pairs to states;*
- $F \subseteq Q$ *is a set of final states (alternatively called accepting states) and*
- $q_s \in Q$ *is the start state.*

An example of a DFA is depicted in Figure 2.1. The DFA is defined by $Q = \{q_1, q_2, q_3, q_4\}, \Sigma = \{a, b, c, d\}, F = \{q_4\}$ and $q_s = q_1$. The start state, the final states and the normal states are respectively colour coded as red, green and orange. Throughout the dissertation an automaton will often be interpreted as a directed graph, with transitions and states referred to as arcs (edges) and nodes, respectively.

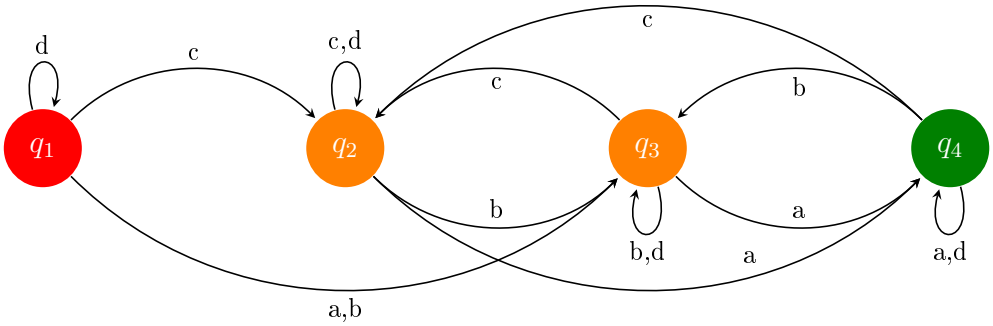

Figure 2.1: A DFA, $\mathcal{D} = (\{q_1, q_2, q_3, q_4\}, \{a, b, c, d\}, \delta, \{q_4\}, q_1)$

Definition 2.11. *A DFA is a* **trie** *iff its transition graph is a tree rooted at the start state* q_s.

For a symbol $a \in \Sigma$ and states $p, q \in Q$ the transition $\delta(p, a) = q$ indicates a mapping from the source state p to a target state q on a symbol a. We use $\perp$ to represent an *invalid* target state of a transition and it is also referred to as an *undefined state*.

Definition 2.12. *A* **complete DFA** *is such that* every *state has a transition on* every *symbol in the alphabet, that is, the transition function* δ *is* total.

Definition 2.13. Total function and Partial function: *A function is a* **total function** *if it is defined for all inputs of the correct type. For example, the output symbol transition's target states for the total deterministic finite automata belong to* Q *for all domain transitions. However, if the function is not defined for some inputs of the right type for some of the domain, then the function is said to be a* **partial function**. *In our case we have partial functions whereby undefined transitions are denoted by* $\perp$. *A total function is denoted* $\rightarrow$ *while a partial function is denoted by* $\nrightarrow$.

Definition 2.14. *Given a complete DFA, the* **Extension of** δ *is defined as the function*

$$\delta^* \in Q \times \Sigma^* \longrightarrow Q$$

where

$$\delta^*(p, \epsilon) = p$$

and for $a \in \Sigma, w \in \Sigma^*, \delta(p, a) = q$

$$\delta^*(p, a.w) = \delta^*(q, w).$$

Definition 2.15. Acceptance of a string by DFA $\mathcal{D}$ *A finite string* w *is accepted by the DFA iff* $\delta^*(q_s, w) \in F$. *Otherwise,* w *is rejected.*

Definition 2.16. Language of a DFA$(\mathcal{D})$ *is defined as* $\mathcal{L}(\mathcal{D}) = \{w \in \Sigma^* |$ w *is accepted by* $\mathcal{D}\}$.

The language of a DFA is the set of accepted strings. An algorithm that uses a DFA to acccept or reject the membership of a string to a language is shown in Algorithm 2.1.

In this dissertation, all algorithms are presented using the Guarded Command Language (GCL) style initiated by Dijkstra [21]. Its use for specification-based algorithm derivation is described in Kourie and Watson [22].

Algorithm 2.1 Test for string membership of a DFA's language

$\{$ **pre** $(\mathcal{D} = (Q, \Sigma, \delta, F, q_s)) \ \wedge \ (x \in \Sigma^*) \ \wedge \ (|x| < \infty)$ $\}$
$y, q := x, q_s;$
$\{$ **invariant :** y is untested and the current state is q $\}$
do $((y \neq \varepsilon) \wedge (\delta(q, head(y)) \neq \perp)) \rightarrow$
$\quad q, y := \delta(q, head(y)), tail(y);$
od;
$\{$ (y is untested and the current state is q) $\wedge$ $((y = \varepsilon) \vee (\delta(q, head(y)) = \perp))$ $\}$
$accept := ((y = \varepsilon) \wedge (q \in F));$
$\{$ **post** $(accept \Leftrightarrow x \in \mathcal{L}(\mathcal{D}))$ $\}$

Definition 2.17. *A* **Failure Deterministic Finite Automaton (FDFA** $\mathcal{F}$**)** *is a six-tuple* $(Q, \Sigma, \delta, \mathfrak{f}, F, q_s)$, *where:*

- $\mathcal{D} = (Q, \Sigma, \delta, F, q_s)$ *is a DFA as defined above; and*
- $\mathfrak{f} \in Q \nrightarrow Q$ *is a (possibly partial) failure function.*

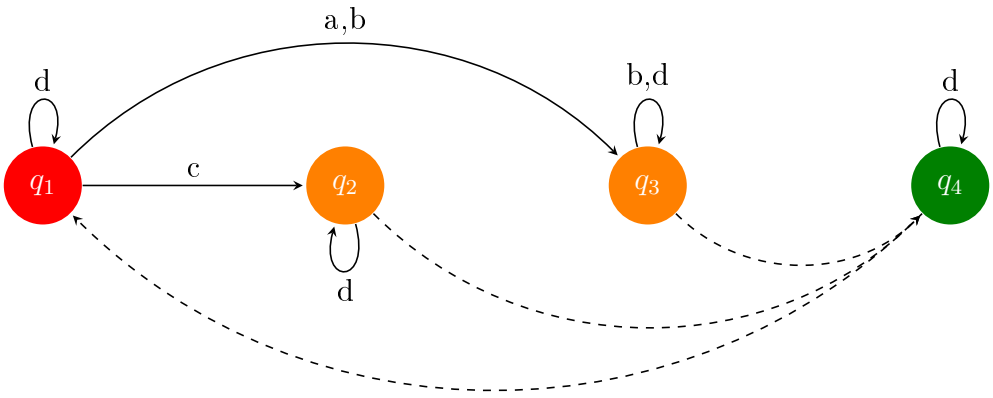

Figure 2.2: An FDFA, $\mathcal{F} = (\{q_1, q_2, q_3, q_4\}, \{a, b, c, d\}, \delta, \mathfrak{f}, \{q_4\}, q_1)$ derived from the DFA in Figure 2.1.

Figure 2.2 shows an FDFA that is derived from the DFA in Figure 2.1. The set of states Q, the alphabet set Σ, the set of accepting (final) states F and the initial state q_s have not been modified from the source DFA. However, notice that some symbol transitions (these are: directed, solid, labelled with letters edges) have been eliminated from the graph. It can also be noted that some dotted, directed and unlabelled arcs have been introduced into the graph. These arcs are the failure transitions, which are the elements of the failure function $\mathfrak{f}$.

The *failure function* $\mathfrak{f}$, simply maps a state p to another state q. This function is denoted by $\mathfrak{f}(p) = q$. The two functions δ and $\mathfrak{f}$ have this relation: if the transition $\delta(p, a)$ is not defined then $\mathfrak{f}(p) = q$ is taken. That is, if the current state has no symbol out-transition associated with the current input symbol, the failure transition is executed and it does not consume the current input symbol.

Definition 2.18. Extension of δ in FDFA: *In the case of an FDFA, δ^* is defined as follows:*

$$\delta^* \in Q \times \Sigma^* \longrightarrow Q \cup \{\perp\}$$

where

$$\delta^*(p, \epsilon) = p$$

and for $a \in \Sigma, w \in \Sigma^*$

$$\delta^*(p, a.w) = \begin{cases} \delta^*(q, w) & \text{if } (\delta(p, a) = q) \ \wedge \ (q \neq \perp) \\ \delta^*(q, a.w) & \text{if } (\delta(p, a) = \perp) \ \wedge \ (\mathfrak{f}(p) = q) \end{cases}$$

Definition 2.19. Acceptance of a string by FDFA $\mathcal{F}$*: An FDFA* $\mathcal{F}$ *is said to accept string* $w \in \Sigma^*$ *iff* $\delta^*(s, w) \in F$*.*

Definition 2.20. Language of an FDFA $\mathcal{F}$ *is defined as* $\mathcal{L}(\mathcal{F}) = \{w \in \Sigma^* |$ w *is accepted by* $\mathcal{F}\}$*.*

An FDFA's language is the failure automata's set of accepted strings. It can easily be shown that every complete DFA has a language-equivalent FDFA and vice-versa. A string processing algorithm for an FDFA extracted from Kourie et al. [11] is depicted in Algorithm 2.2.

Algorithm 2.2 Test for string membership of an FDFA's language

{ **pre** $(\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, q_s)) \ \wedge \ (x \in \Sigma^*) \ \wedge \ (|x| < \infty)$ }
$y, q := y, q_s$;
{ **invariant:** y is untested and the current state is q }
do $(y \neq \varepsilon) \wedge (\delta(q, head(y)) \neq \perp) \ \rightarrow \ q, y := \delta(q, head(y)), tail(y)$;
[] $(y \neq \varepsilon) \wedge ((\delta(q, head(y)) = \perp) \ \wedge \ (\mathfrak{f}(q) \neq \perp)) \ \rightarrow \ q := \mathfrak{f}(q)$;
od;
{ y is untested and the current state is q
$\wedge ((y = \varepsilon) \vee ((\delta(q, head(y)) = \perp) \wedge (\mathfrak{f}(q) = \perp)))$ }
$accept := ((y = \varepsilon) \wedge (q \in F))$;
{ **post** $(accept \Leftrightarrow x \in \mathcal{L}(\mathcal{F}))$ }

Definitions and notations that relates to failure functions are presented below.

Definition 2.21. *A* **failure path** *is a consecutive series of FDFA states* $\langle p_0, p_1, \ldots p_n \rangle$ *from* p_0 *to* p_n *such that* $\forall i : [0, n) \cdot \mathfrak{f}(p_i) = p_{i+1}$ *. It is denoted*

$p_0 \overset{f}{\rightsquigarrow} p_n$.

Definition 2.22. *A* **failure alphabet of a failure path**. *If* $p_0 \overset{f}{\rightsquigarrow} p_n$ *is a failure path, then we use* $\Sigma_{p_0 \overset{f}{\rightsquigarrow} p_n}$ *to denote its failure alphabet* $\Sigma_{p_0} \cap \Sigma_{p_1} \cap \cdots \cap \Sigma_{p_n}$. *Note that for* $i = 0, \ldots n$, $\Sigma_{p_i} \subseteq \Sigma$ *is the set of symbols that do not have outgoing symbol transitions from a state* p_i.

Definition 2.23. *A* **failure cycle** *is a connected failure path, that is* $p_j \overset{f}{\rightsquigarrow} p_j$.

Definition 2.24. *A* **divergent failure cycle** *is a cycle whose failure alphabet is non-empty.*

Divergent cycles must be avoided when constructing FDFAs. A divergent cycle may be disastrous during string membership processing since it may lead to an infinite sequence of failure states being traversed. This is because a symbol in its failure alphabet will indefinitely not be consumed while processing in the failure cycle.

Further information about DFAs and FDFAs is beyond the scope of this dissertation and can be found elsewhere in the literature. Examples are: [2, 11, 23, 24].

2.1.3 Formal Concept Analysis

Formal Concept Analysis (FCA) is a mathematical theory of data analysis. Some data sourced from a collection of objects and their characteristics are organized into a meaningful hierarchy of information. The term formal concept analysis was coined by Rudolf Wille in 1984. There are many publications from the FCA community which provide related basic definitions. (See [25, 26].) We introduce FCA here because it is a technique that is applied in this research. Further applications of FCA in stringology are summarized in Kourie et al. [27]. Below we provide FCA definitions which relate to the scope of this dissertation.

Definition 2.25. *A* **formal context** *is the triple* (O, A, I), *where* O *is a set*

of objects, A is a set of attributes possessed by the objects and $I \subseteq O \times A$.

Thus, the relation I indicates which objects possess which attributes in the particular context. For example, if $\langle o_i, a_j \rangle \in I$, then this means that object o_i has the attribute a_j. In general, objects refer to discrete entities in any domain that are described by discrete attributes. This dissertation is concerned with finite automata, and so the objects and attributes will be in reference to this domain.

A formal context may be represented by a cross-table. Table 2.2 below is an example of such a cross-table. It has been constructed to represent information about a complete DFA. Here is how we build a formal context from a complete DFA. For a given symbol, $a \in \Sigma$, we identify all states $p's \in Q$, with the same transition value (target state, $q \in Q$). An attribute in the formal context becomes the combination of the symbol and the destination state while its objects are the source states. For a given state, $q \in Q$, its attributes are also known as its *abilities*, a term coined by Björklund et al. [10].

To illustrate how a formal context can be derived from a complete DFA in the manner described above, consider the transition table of a DFA as shown in Table 2.1. This table represents the arcs of the DFA depicted in Figure 2.1. For example, since the entry in the second row (for state q_2) and third column (for alphabet symbol c) is q_3, we infer that there is an arc from q_2 to q_3 labelled by c.

Table 2.2 then gives the corresponding formal context for this DFA. The rows represent the objects, in this case the states of the DFA. The columns represent attributes of the context, in this case the ⟨symbol, destination⟩ pairs for arcs leaving the object (state) in the given row. The first row of Table 2.2 shows, for example, that state q_1 has a transition on a to q_1, a transition on b to q_2, a transition on c to q_3 and a transition on d to q_1.

Such a formal context constructed from a given DFA will henceforth be called the DFA's *state/out-transition context*.

Table 2.1: A DFA transition table sourced from Figure 2.1

	a	b	c	d
q_1	q_1	q_2	q_3	q_1
q_2	q_1	q_2	q_3	q_2
q_3	q_1	q_2	q_3	q_3
q_4	q_2	q_2	q_3	q_4

Table 2.2: The State/out-transition context for the DFA from (Table 2.1)

	q_1, a	q_2, a	q_2, b	q_3, c	q_1, d	q_2, d	q_3, d	q_4, d
q_1	✓		✓	✓	✓			
q_2	✓		✓	✓		✓		
q_3	✓		✓	✓			✓	
q_4		✓	✓	✓				✓

Note that an upper bound on the possible number of cells of a state/out-transition context is $|Q|^2 \times |\Sigma|$ — i.e. a row for every state (i.e. $|Q|$ rows) and maximally $|Q| \times |\Sigma|$ columns for different attributes. Because we are considering here only complete DFAs, a given object (state in a row) will always have exactly $|\Sigma|$ attributes (out-transitions).

The minimum number of cells of a state/out-transition context is $|Q| \times |\Sigma|$, which will be achieved when the out-transitions on each symbol at each state end at the same target state, i.e. when

$$\forall a \in \Sigma \wedge \forall q, p \in Q : \delta(a, q) = \delta(a, p).$$

Definition 2.26. *A* **formal concept** *c of a formal context (O, A, I) is a pair $\langle X, Y \rangle$, where $X \subseteq O$ and $Y \subseteq A$ such that:*

- $(X \times Y) \subseteq I$;
- *Y is the largest subset of A such that $(X \times Y) \subseteq I$ and*
- *X is the largest subset of O such that $(X \times Y) \subseteq I$.*

The set X is called the *extent* of the concept (denoted by *ext(c)*), and the set Y is called the concept's *intent* (denoted by *int(c)*). Note that the extent and intent of a concept are related in a very particular way. A concept's extent is maximal in the sense that objects in that set hold in common *exactly* the attributes in concept's intent — there is no object outside of the extent that also has those attributes as a subset of its attributes. Additionally, the concept's intent is maximal: if an attribute is not in the concept's intent then at least one of the objects in the extent does not possess this attribute.

In formal concept analysis, concepts are hierarchically related by a partial ordering. The partial order is defined as follows.

Property 2.1.1. The partial order on concepts: *Given two concepts c_i and c_j of a context, if $ext(c_i) \subseteq ext(c_j)$ then $c_i \leq c_j$. (Equality holds if and only if $i = j$) (Refer to van der Merwe et al. [28].).*

By definition of a concept, if $c_i \leq c_j$ holds then it is also the case that $int(c_j) \subseteq int(c_i)$.

Definition 2.27. *A* **formal concept lattice** *is the set of formal concepts derived from a formal context whose ordering is defined in Property 2.1.1.*

Where clear from the context, the terms "lattice" or "concept lattice" should be taken to mean a formal concept lattice. As it will be illustrated later, a lattice is often visually represented by a so-called *Hasse* diagram.

Formal concept analysis based on formal concept lattices is an active research and application area in the field of knowledge base intelligence. Lattices are used to organize data in data mining and knowledge discovery applications, for example: Zhou et al. [29]. In Kourie et al. [27] an overview is given of where FCA has been applied in stringology. The overview includes a summary of a multiple keyword pattern matching algorithm that uses FCA, which was proposed by Venter et al. [30].

In this work we apply concept lattices in stringology by using the state/out-

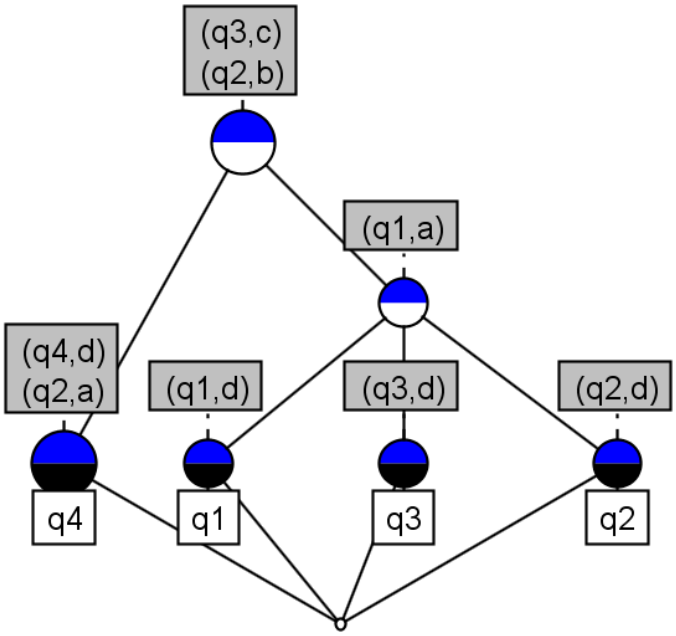

Figure 2.3: A formal concept lattice derived out of the state/out-transition in Figure 2.2

transition context from a *complete* DFA as input for the so-called DFA Homomorphic Algorithm (DHA) proposed in Kourie et al. [9], in order to derive a language-equivalent FDFA of the DFA. As explained below, the state/out-transition context is used as input to a lattice construction algorithm that returns a formal concept lattice. This will be called a *state/out-transition lattice*. The Hasse digram (or line diagram) depicting the state/out-transition lattice associated with the state/out-transition context shown in Table 2.2 is given in Figure 2.3.

In Figure 2.3 the concepts (i.e. nodes) are circles labelled attached with labeled grey boxes and white labelled boxes. Each node contains either a combination of blue-and-black colors or blue-and-white colors. A black color represent a concept's *own objects*, a white color denotes a concept's *derived objects* and a blue color stand for a concept's *own attributes*. The own objects of a concept are objects that belongs to the current concept and no other concept at a lower level contains any of those objects. Derived objects of

a concept are represented as a set of objects that share some attributes, they are derived from objects of concepts at lower levels of the graph. The own attributes of a concept are attributes of which none of its elements are shared with any other parent (higher level) concept. From the graph, own objects caption are the white boxes while own attributes are labels of the grey boxes. The top node represent concepts where all the states (objects) have shared attributes while the bottom node represents all the attributes that are exclusive to a set of states.

Figure 2.3 provides elaborative details of the association of states and state/out-transitions from the lattice diagram. The diagram is read from bottom-up. At the bottom root node, none of the state/out-transitions is exclusive to any state. In other words, there is no single object that contains all the attributes. Moving one level up, there is a concept that has one own object $\{q_1\}$ and possesses a single own attribute $\{(q_1, d)\}$. It has to be noted that other elements of the intent for the above stated concept include all its parent attributes (i.e. (q_1, a)) and ancestor attributes (i.e. (q_2, b) and (q_3, c)). Similarly, the concept nodes that have own object/state $\{q_2\}$ and $\{q_3\}$, catagorically have $\{(q_2, d)\}$ and $\{(q_3, d)\}$ as own attributes. Their other attributes are contained in connected higher level concepts. Meanwhile, for the concept with own object: $\{q_4\}$, there are two own attributes (q_4, d) and (q_2, a) and two inherited attributes: (q_2, b) and (q_3, c). Going up to the second layer of the latticce diagram, there is a concept that consists of a set of states $\{q_1, q_2, q_3\}$ with an own attribute (q_1, a) and parent attributes. Lastly, the top level concept has an extent of $\{q_1, q_2, q_3, q_4\}$ and exclusively has the intent $\{(q_3, c), (q_2, b)\}$.

Each concept in a state/out-transition lattice is characterised by a certain value, called its arc-redundancy, which is defined as follows:

Property 2.1.2. Arc Redundancy (AR) *for a concept c is an integer value, $ar(c) = (|int(c)| - 1) \times (|ext(c)| - 1)$.*

In Kourie et al. [9] it is shown that a non-negative arc redundancy value of a concept represents the number of arcs that may be reduced by doing the

following:

1. singling out one of the states in the concept's extent;
2. at all the remaining states in the concept's extent, removing all out-transitions mentioned in the concept's intent;
3. inserting a failure arc from each of the states in step 2 to the singled out state in step 1

The expression, $|ext(c)| - 1$ represents the number of states in step 2 above. At each such state, $|int(c)|$ symbol transitions are removed and a failure arc is inserted. Thus, $|int(c)| - 1$ is the total number of transitions removed at each of $|ext(c)| - 1$ states so that $ar(c)$ is indeed the total number of arcs saved by the above transformation.

From Figure 2.3, the arc redundacy of the top node concept is 3 as its $|ext(c)| = 4$ and its $int(c)| = 2$. And, for the second level concept, as $|ext(c)| = 3$ and $|int(c)| = 3$ then $AR = (3-1) \times (3-1) = 4$. The remaining concepts have $AR = 0$.

Definition 2.28. The Positive Arc Redundancy Set (PAR set) *is the set of concepts in a state/out-transition lattice that have arc redundancies greater than 0.*

The PAR set excludes all concepts with a negative or zero arc redundancy. From the example provided in Figure 2.3 only the two concepts at the two higher levels qualify to be elements of the PAR set.

For a given PAR set, the concept with the maximum arc redundancy value is denoted by $maxAR$, the concept with the maximum intent size is denoted by *maxIntent* and the concept with the minimum extent size is denoted by *minExtent*.

2.2 Background and Related Work

This section presents the background and literature review of failure-DFAs. Firstly, the Aho-Corasick algorithms are introduced, because they play an important role in this research. The section then goes on to highlight other literature sources that have investigated FDFAs. And finally, it discusses the application of FDFAs in compiler construction.

A language that is accepted by a DFA is called a regular language. The regular language is the simplest class of languages from the Chomsky hierarchy, see: [31, Chapter 11], [32], [33, p. 281]. The Chomsky hierarchy ranks languages ordered from the most restrictive to the most general: regular, context-free, context-sensitive, and recursively enumerable. It is well known that any finite set of strings from an alphabet is a regular language, and can therefore be represented by a DFA. String processing that is based on a DFA is generally somewhat simpler and more efficient than string processing of languages higher up in the Chomsky hierarchy.

Suppose, then, that P is a finite set of N strings of finite length drawn from the alphabet Σ. This set is therefore a regular language. It can therefore be represented by a DFA and, in this case, the transition graph is a tree (referred to as a *trie* in the DFA context). As an example of a trie, consider $P = \{he, her, him, she\}$. The corresponding trie for this set of strings is shown in Figure 2.4. Notice that it is characterised by the fact that each state (other than the start state) has exactly one inbound transition. Every path from start state to some other state spells out a prefix of one of the keywords in P. As a result every state can be identified with the proper prefix of the path that ends at that state. As a corollary to this observation, an internal state will be a final state if this prefix is a full keyword in P. It has to be noted too that all the trie's leaves are final states.

Such a trie may therefore be used to represent a finite set of finite-length keywords in a multi-keyword exact pattern matching task. In such a task, a text string, $T \in \Sigma^+$ is given, and all indices in the text are sought at

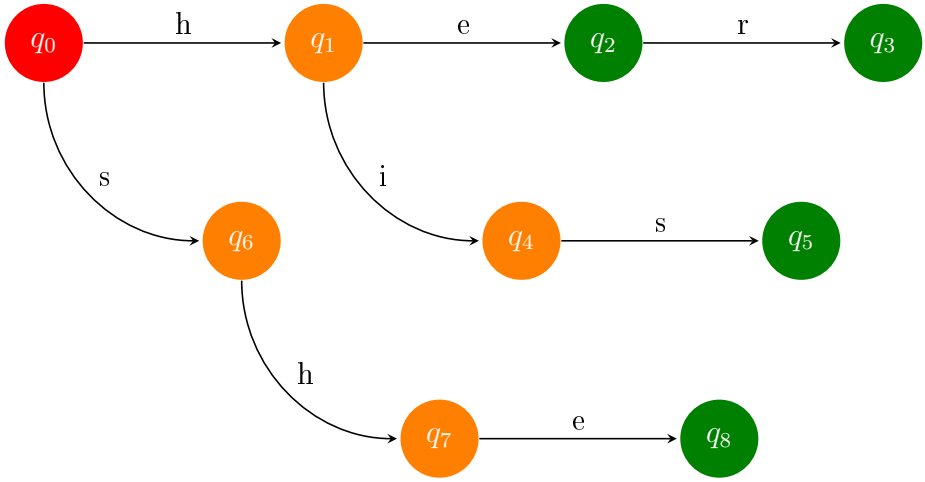

Figure 2.4: An example of a trie with keyword set $P = \{he, her, his, she\}$

which patterns in the keyword set occur. In principle, Algorithm 2.1 can be adapted to use T and the trie for P as input to indicate whenever a final state is reached that the associated keyword of P has been detected in T. However, in order to do this, the scanning of T should not terminate if no progress can be made in the trie. Instead, transitions should be inserted into the trie to appropriate states so that the remaining part of T can be scanned.

The task in multi-keyword exact pattern matching can therefore be seen as finding a DFA whose language is $\Sigma^* P$. One type of the Aho-Corasick (AC) algorithm proposed by Aho and Corasick [13] does precisely that, where the DFA constructed is minimal. Each final state of this DFA corresponds with one of the patterns in P. This AC variant then constructs a complete DFA that has an arc from each final state going back to the start state on symbols that do not lead to a prefix of one of the patterns. This DFA is called the *AC-opt* DFA. The "opt" stands for optimal. It is optimal because the DFA's string processing algorithms (such as Algorithm 2.1) that are used for scanning the text need not have to contain the additional logic of FDFA string processing algorithms (refer to Algorithm 2.2) have in order to determine whether or not to take a failure transition. However, it is not optimal in terms of memory requirements, since it is a complete DFA and therefore has to provide storage

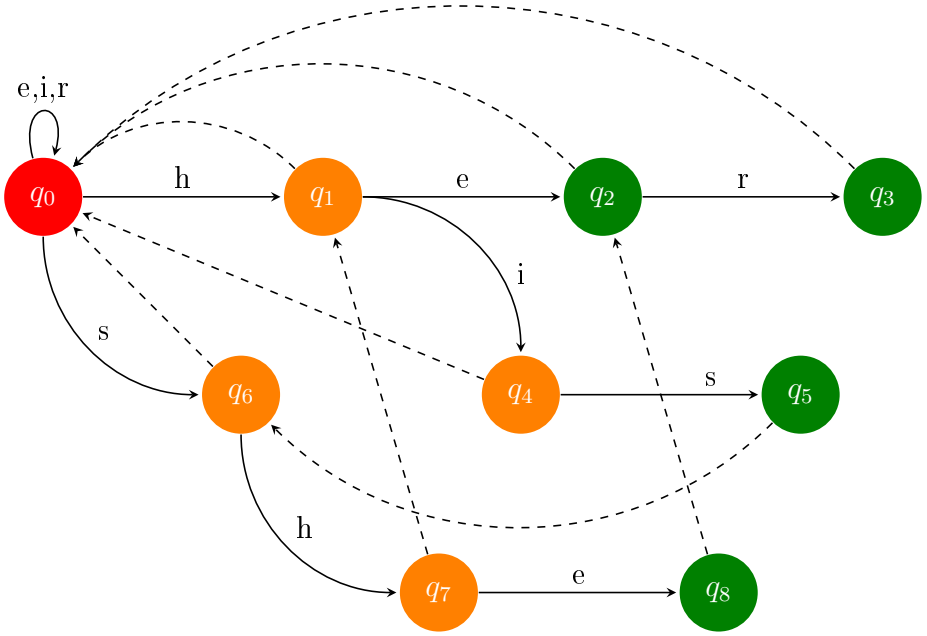

Figure 2.5: An AC-fail automaton for the keyword set $P = \{he, her, his, she\}$

to record a transition from every state on every alphabet symbol.

The second version of AC algorithm constructs an FDFA that is language-equivalent to the previously mentioned DFA. However, the FDFA is not constructed from the minimal AC DFA mentioned in the previous paragraph. Instead, it first constructs a trie (which, as discussed above, is itself a DFA whose language is exactly the finite language P). And then it inserts failure arcs using the "prefix of a suffix" [23, Chapter 4] approach — that is, fail from a suffix s of a word $u.s$ that is a longest prefix of another word $s.v$ in the trie. The Aho-Corasick failure automaton has the property of introducing the maximum number of failure arcs, and it is called *AC-fail* FDFA. Note that the word $s.v$ could be any word in P including $u.s$. We now further illustrate the notion of "prefix of a suffix" by an example visualized in Figure 2.5. Suppose p is a state in the trie representing the string and keyword *she* and suppose q is another state representing the prefix string *he* of the keyword *hers*. Then a transition from p to q would indicate that *he* is the longest suffix of *she* that matches a prefix of some other keyword. An example of an

AC-fail automaton is depicted in Figure 2.5.

The construction time for an AC-fail automaton is $O(n \log |\Sigma|)$, where n is the number of words in the pattern set and Σ is the alphabet set [34]. More details about the construction of an AC-fail automaton may be found in [13, 23, 18, 35]. Dori and Landau [34] mentioned that the runtime complexity of multiple keyword pattern matching algorithms that make use of AC-fail FDFA is $O(m \log |\Sigma| + k)$, where k is the number of all matching occurrences of keywords in the input character sequence T, and m is the size of the input text string, T. Björklund et al. [24] stated that the storage requirements of a failure automaton are proportional to the total number of transitions, and hence the need for a maximum transition reducing FDFA.

Several authors have proposed approaches to improve Aho-Corasick automata's execution time through efficient hardware architectures. Examples include: [36, 12]. Watson [23] classified classical pattern matching algorithms (including AC) into a taxonomy, and implemented a toolkit. Subsequently, this toolkit was improved by Cleophas and Watson [37].

There is a body literature about FDFAs that are not AC-fail FDFAs. The pioneers are Knuth et al. [38] who developed the Knuth-Morris-Pratt (KMP) algorithm—a single-keyword exact pattern matching algorithm. The Aho-Corasick algorithm generalized the KMP algorithm from single to multiple keyword matching. Crochemore and Hancart [2] provided an overview of various FDFA related research that has been conducted in specific contexts. However, Kourie et al. [9] described a *general algorithm* for generating an FDFA from any DFA. This generalises the AC algorithm which can only construct an FDFA for a finite language (i.e. a finite set of patterns of finite length).

An alternative approach to constructing FDFAs from an arbitrary DFAs is given by the *Delayed-input DFA (simply called, D^2FA)* algorithms developed by Kumar et al. [12]. This approach will be discussed in the succeeding section.

Björklund et al. [24, 10] presented a couple of theoretical FDFA contri-

butions. The first contribution is a proof that both the failure reduction problem and the transition minimization problem are NP-complete. The transition reduction problem is the construction of an FDFA by removing some symbol transitions while adding failure transitions. Note that there are many resulting DHA FDFA solutions from a given input DFA, we are merely trying to obtain failure automata that can have relatively high transition reduction. The transition minimization problem is the same as the former but it has an extra property. The extra property is that additional states may be included into the FDFA when required as long as the language of the input DFA is preserved. This property allows the FDFA to have more states than the language equivalent DFA. In their second contribution, the authors sought for different ways to trace transition reduction. This was conducted by providing an algorithm which demonstrates that the failure reduction problem can be estimated at least two-thirds factor of the transition size when compared with an optimal algorithm.

Recently Cleophas et al. [39] used ideas inspired by Kourie et al. [9] to modify factor oracles (FO) constructing algorithms to introduce failure transitions and therefore generating failure factor oracles (FFO). A factor oracle is a data structure that recognizes all factors of a single keyword. By introducing failure arcs, the empirical results presented showed that up to 9% failure transitions were saved. FOs have a compact and efficient representation. Short natural language (English) words of length of up to 14 characters were tested. Improvements on the FO algorithms were addressed in Cleophas et al. [40]. The improved version of (F)FOs removes potential failure cycles through transition minimization methods and optimization by partial memoization techiques for larger input strings T to be processed by the FFs and the FFOs. And, the (F)FO were tested with pattern matching on DNA sequences in this case. Lastly in [41], the authors included the FFO and FO algorithms into the pattern matching taxonomies of [37, 23]. Moreover, the authors compared the performances of FFO and FO in pattern matching and found that FO performed better.

The application of failure transitions in the context of subsequence au-

tomata has recently been discussed in Bille et al. [42]. A subsequence of a finite text string s is any string that is obtained from deleting zero or more characters from s. A subsequence automaton is the minimal deterministic finite automaton accepting all subsequences of s. The authors demonstrated that by using failure transitions, much smaller subsequence automata may be formed. Moreover, the authors compared the performances of FFO and FO in pattern matching and found that FO performed better.

One area of application for such DFAs and FDFAs is lexical analysis in compiler construction. A lexical analyser in a compiler is essentially the software module that uses DFA technology to verify that the input program's — such as variable names, keywords, numbers, etc — conform to the language's specification and then outputs a string of tokens to be used by other compiler modules. Software such as *Lex* (and *Flex*, a faster Lex) for C/C++ which was developed by Lesk and Schmidt [43] and Java's *JFlex* (see Klein et al. [44]) have been developed to generate the code for a lexical analyser from a given program.

2.3 A Description of the D^2FA Algorithm

Kumar et al. [12] proposed an approach to change a DFA into an FDFA in 2006, prior to the publication of the DHA approach described by Kourie et al. [9]. This so-called Delayed-input Deterministic Finite Automata (D^2FA) algorithm had not been discovered by the authors of the DHA algorithm because it was not published in the mainstream stringology literature sources, but in literature relating to network security. Additionally, the term "delayed-input" DFA (or D^2FA) had been independently invented by the authors of the algorithm, and consequently did not show up under literature searches relating to "failure" DFAs. I discovered the literature describing this algorithm quite late after carrying out the research for this dissertation. A minor contribution of my research is therefore the bringing together of these two avenues of research to solve the same problem that had until now been

conducted independently. The algorithm has two variants and it is briefly described below. The first of these variants was implemented and will be included throughout this study. In this text, it will be referred to as the D^2FA algorithm.

In order to derive a failure automaton from a DFA, an undirected graph called the *space reduction graph* is constructed. Its nodes are defined by the states of the DFA. Each edge connecting two states (q_i and q_j) is assigned a *weight* which is the sum of all transitions such that $\delta(a, q_i) = \delta(a, q_j)$ minus 1 (and $q_i, q_j \in Q$ and $a \in \Sigma$). Meanwhile, all but one symbol transitions with the property $\delta(a, q_i) = \delta(a, q_j)$ are removed.

Failure transitions are determined as follows. Firstly, out of the above discussed space reduction graph, a maximal weight spanning tree is generated. The D^2FA algorithm builds a directed spanning tree to incrementally generate failure transitions (the authors referred them as *default transitions*). An algorithm by Kruskal [45] was used to generated the directed spanning tree from the weighted space reduction graph.

Next, the algorithm arbitrarily selects a root node (state) for the spanning tree. Then, for each edge of the spanning tree connecting two nodes/states, that edge becomes a failure transition. For two connected nodes of the spanning tree q_i and q_j, a failure transition is created out of this spanning tree node link such that the failure transition is directed towards the state that has the least depth (i.e. the least number of edges from the selected root node). It should be noted that under no circumstances does the FDFA produced by the D^2FA algorithm have divergent failure cycles, as all failure arcs are directed towards the root state.

Two variants of the algorithm were proposed. The first one is based on the original maximal spanning tree and the other on a redefined maximal spanning tree. The latter, allows tree nodes to be fairly distributed from the root by setting all the nodes to be at a controlled diameter[1] from the root

[1]A diameter (also known as a *width*) of a tree is the number of edges with the longest path between two leaf nodes in the tree.

node. A small diameter helps in limiting the number of failure transitions taken until a symbol is consumed by a symbol transition. With an aid of hardware implimentation (for faster processing), the reduction on transitions obtained reached 95%. An improvement to replace some failure transitions by specific symbol transitions on D^2FA was proposed by Ficara et al. [46].

Note that there is the following connection between the weights used in the space reduction graph and the notion of "arc redundancy" described earlier. It can easily be shown that if a concept in a state/out-transition lattice has exactly two states in its extent, then the arc redundancy of that concept will be the same as the weight assigned to the arc connecting those two states in the space reduction graph.

2.4 Conclusion

In the current chapter definitions, notations and notes related to these three research areas; stringology, finite automata and formal concept analysis were presented. A background study of failure automata and other related works were presented and discussed. Notably, a kind of FDFA called D^2FA was singled out for a deeper discussion because it will be highly referenced in the forthcoming chapters. The next chapter provide detailed description of the DFA-homomorphic algorithm which is core algorithm that is studied by this dissertation.

Chapter 3

The Transformation of DHA

This research seeks to to examine how effective will various algorithms derive FDFAs from complete DFAs. Our early trials of the original *DFA-homomorphic* algorithm (DHA) using some heuristics so-called MaxArcRedundancy as suggested by Kourie et al. [9] showed that the algorithm did not produce FDFAs that remove the most possible symbol transitions out of input DFAs. Here, the initial DHA is tuned into producing FDFAs that are possibly the best transition savers. Four concrete variants of DHA are proposed with a view to examining how effectively they produce FDFAs with as small a number of transitions as possible. This chapter provides a detailed description of these variants of DHA.

3.1 A Description of the DHA

For DHA to convert a DFA into an FDFA (as depicted in Figure 3.1), a three stage transformation of the DFA has to be undertaken. Initially, the DFA is represented as a state/out-transition formal context. Such a formal context is the required standard input for formal concepts generating software. The software for building formal concepts is called Formal Concept Analysis Research Toolbox (FCART). FCART has been used in this research to build

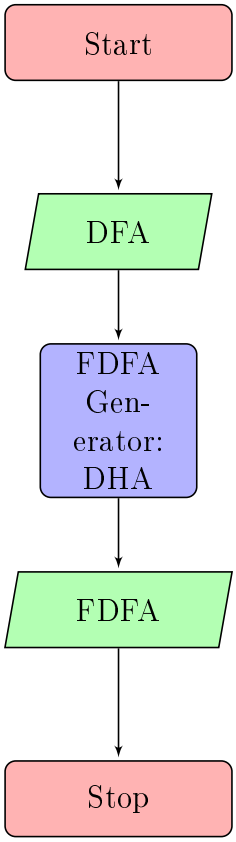

Figure 3.1: The DHA's DFA to FDFA Conversion Process

state/out-transition concepts from the provided context. This package will be described in the Chapter 4. From the built concepts, the set of positive arc redundancy concepts (i.e. PAR Set) is extracted to serve as input for a variant of the DHA, which then provides an FDFA that is language-equivalent to the original DFA. This sequence of conversions is illustrated in a nutshell in Figure 3.2.

The basic DHA proposed in Kourie et al. [9] is outlined in Algorithm 3.1. Algorithm 3.1 is summarised in the next couple paragraphs.

The variable O is used to keep track of states that are not the source of any failure transitions. This is to ensure that a state is never the source of more than one failure transition. Initially all states are elements of O. PAR is used to store the set of concepts with positive arc redundancy. The

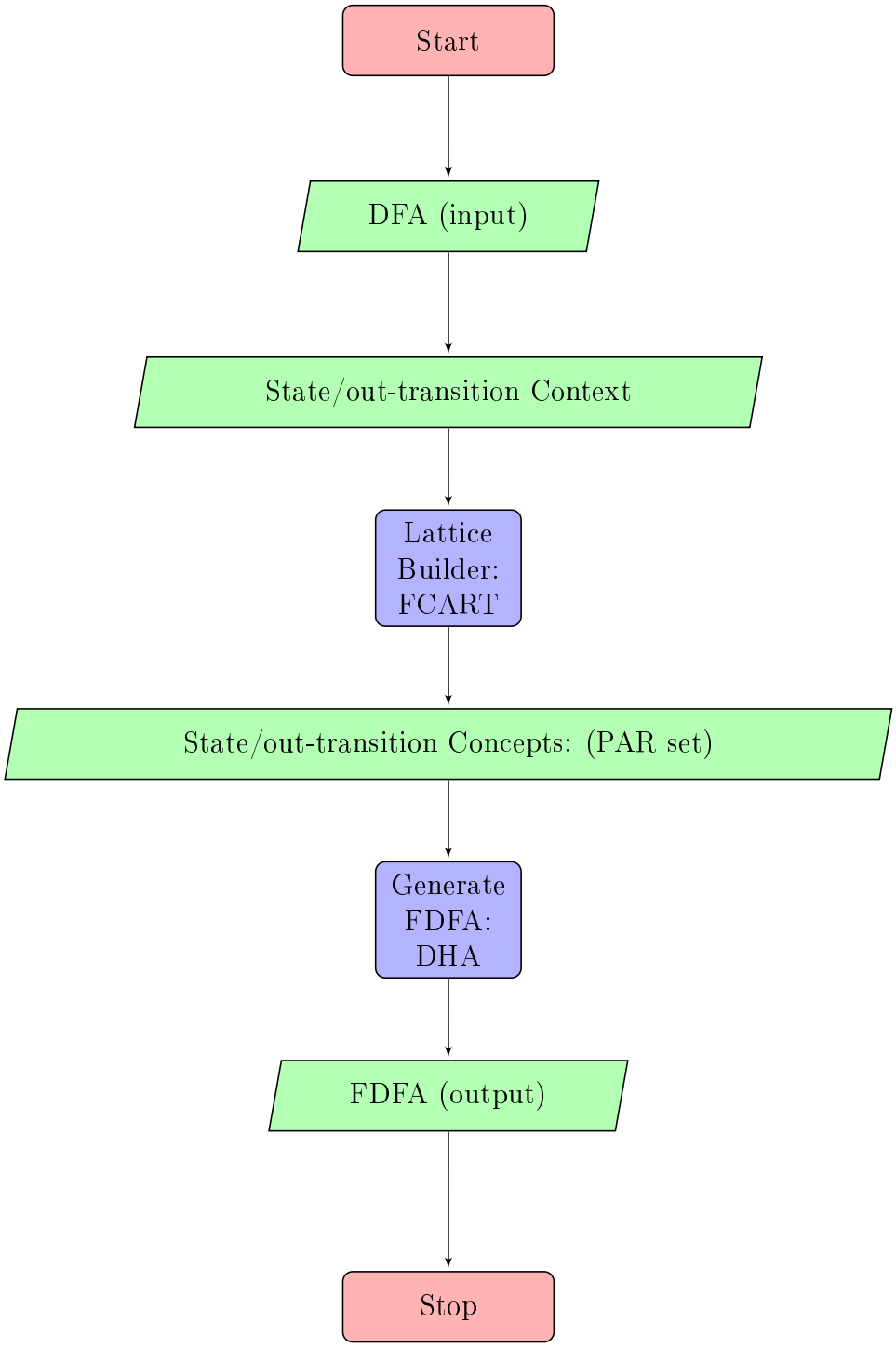

Figure 3.2: The DHA based DFA-to-FDFA construction in detail

following sequence of actions are taken for as long as O and PAR is not empty.

Algorithm 3.1 The Original DFA-Homomorphic Algorithm

$\{$ **pre** $(\mathcal{D} = (Q, \Sigma, \delta, F, q_s)) \ \wedge \ (O \neq \emptyset) \ \wedge \ (PAR \neq \emptyset) \ \}$
$O := Q$; $\{$ States from which failure transition may start $\}$
$PAR := \{c | ar(c) > 0\}$; $\{$ Set of concepts with $ar(c) > 0$ $\}$
do $((O \neq \emptyset) \wedge (PAR \neq \emptyset)) \rightarrow$
  $c = SelectConcept(PAR)$;
  $PAR := PAR \backslash \{c\}$;
  $t = getAnyState(ext(c))$;
  $ext'(c) := ext(c) \backslash \{t\}$;
  for each $(s \in ext'(c) \cap O) \rightarrow$
    if a divergent failure cycle is not created $\rightarrow$
      for each $((a, r) \in int(c)) \rightarrow$
        $\delta := \delta \setminus \{\langle s, a, r \rangle\}$
      rof;
      $\mathfrak{f}(s), O := t, O \backslash \{s\}$;
    fi
  rof
od
$\{$ **post** $(\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, q_s)) \ \wedge \ ((O = \emptyset) \ \vee \ (PAR = \emptyset)) \ \}$

A concept c is selected from the PAR set for consideration. It is then removed from the PAR set so that c is no longer available in subsequent iterations. The specification given in Algorithm 3.1 leaves open how this choice will be made. The authors' initial version proposed specifically selecting a concept, c, with maximum arc redundancy. This was based on the conjecture that such a "greedy" selection would result in an FDFA with significantly fewer transitions than the original input DFA. However, the conjecture was never confirmed with real data.

From c's extent, one of the states, t, is chosen to be a failure transition target state. Again, the initial version of DHA is nondeterministic on a criterion for which a state in $ext(c)$ can be selected. The selected state t is

removed from $ext(c)$ and the remaining set of states is called $ext'(c)$. Then, for each state s in $ext'(c)$) that qualifies to be the source of a failure transition (i.e. that is also in O), the following action is taken.

Firstly, a check is made to verify that installing a failure arc from s to t will not generate a divergent failure cycle. If it will, then s is ignored and the next candidate state to be a possible source of a failure transition is examined. If it is assured that a failure arc from s to t will not generate a divergent failure cycle then the following actions are taken.

The transition function δ is updated so that all transitions exiting from s that are mentioned in $int(c)$ are removed. A failure transition is then installed from s to t. Because state s has become a failure transition source state whose target state is t, it may no longer be the source of any other failure transition, and so is removed from O.

Repeatedly, these steps are carried out until it is no longer possible to install any more failure transitions. Further details about the original DHA are available in Kourie et al. [9, 11].

It should be noted that in this particular formulation of the abstract algorithm, the PAR set is not recomputed to reflect changes in arc redundancy as the DFA is progressively transformed into an FDFA as suggested in Kourie et al. [11]. This does not affect the correctness of the algorithm, but may affect its effectiveness in reducing the overall number of transitions in the resulting FDFA. Investigating such effects is not within the scope of this study.

3.2 Modifying DHA

Three changes to the DHA shown in Algorithm 3.1 have been applied to adapt the algorithm so that it can be concretely implemented. Some of these modifications were done as part of this research, while others were suggested in [9]. Two modifications relate to the non-specific fourth and sixth lines of the pseudo-code in original DHA version, Algorithm 3.1, namely the

lines:

- $c := selectConcept(PAR)$; and
- $t := getAnyState(ext(c))$;

respectively. These changes are effected to address the algorithm's challenges of non-determinism in choosing a concept and in selecting the target state of the failure transition from the chosen concept's extent.

Algorithm 3.2 The Modified DHA

$\{$ **pre** $(\mathcal{D} = (Q, \Sigma, \delta, F, q_s)) \ \wedge \ (O \neq \emptyset) \ \wedge \ (PAR \neq \emptyset)$ $\}$
$PAR := \{c \mid ar(c) > 0\}$;
$O := Q \backslash q_s$;
do $((O \neq \emptyset) \wedge (PAR \neq \emptyset)) \rightarrow$
  $c := MaxARConcept(PAR) \vee$
    $MinExtentConcept(PAR) \vee$
    $MaxIntentConcept(PAR) \vee$
    $MaxIntMaxExtConcept(PAR)$;
  $PAR := PAR \backslash \{c\}$
  $t := ClosestToRoot(c)$
  $ext'(c) := ext(c) \backslash \{t\}$;
  for each $(s \in ext'(c) \cap O) \rightarrow$
    if a divergent failure cycle is not created $\rightarrow$
      for each $((a, r) \in int(c)) \rightarrow$
        $\delta := \delta \setminus \{\langle s, a, r \rangle\}$
      rof;
      $\mathfrak{f}(s), O := t, O \backslash \{s\}$;
    fi
  rof
od
$\{$ **post** $(\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, q_s)) \ \wedge \ ((O = \emptyset) \ \vee \ (PAR = \emptyset))$ $\}$

As a consequence of the non-determinism in choosing a concept, we propose four variants of the algorithm. Each one involves a different greedy[1]

[1]By greedy we mean that an element from the set is selected based on some maximal or minimal feature, without regard to possible opportunities lost in the forthcoming iterations

criterion for choosing concept c from PAR set. This is expressed in Algorithm 3.2 as the assignment to c to the disjunction of four different possible calls to differently coloured functions parameterised by PAR. However, this should not be regarded as a strict boolean operation and assignment, but rather as a loose but compact way of saying that each variant uses a different function to determine which element from PAR should be assigned to c.

In addition to these four alternative heuristics, a specific criterion is proposed in the fifth line for choosing the target state, t for the failure transitions. It will be seen below that this criterion also has two variants. Lastly, in the second line $O := Q$, a modification is made to reduce the execution time of the algorithm. These changes are depicted as coloured assignment statements in Algorithm 3.2. These modifications are explained in the subsections that follow.

3.2.1 Select a Concept c from PAR

Firstly, we look at the greedy approaches that are used to select concepts out of the positive arc redundancy set. All the heuristics used can, in principle, affect the shape (transition-wise) of the FDFA created. The research aims to find heuristics that remove the most symbol transitions from the DFA. The *MaxARConcept* criterion was by Kourie et al. [9], early experimental trials showed that it was not the best approach in reducing transitions of a DFA, therefore, we proposed the three remaining approaches.

- $c := MaxARConcept(PAR)$;
  This is the initial criterion proposed in Kourie et al. [9], whereby a concept with the maximum arc redundancy is preferred. It is a greedy strategy that maximises the net change of transition size per iteration. (That is, delete maximum regular transitions and insert maximum failure transitions per iteration.) Recall from the definition of AR, this

by making these selections.

strategy targets concepts that have a middle-ground between the number of potential symbol transitions can be removed from a DFA and the number of possible failure transitions that can be inserted. Preliminary trials with this criterion did not yield impressive transition reduction results. That suggested that alternative criteria might be more effective and gave rise to the subsequent proposals.

To facilitate the implementation of this criterion, the concepts of the state/out-transition lattice may be stored in a list data structure in which elements are sorted in descending order of their arc redundancy values. Each successive selection of a concept from the list is merely taken from the head of the list.

- $c := MaxIntentConcept(PAR)$;
  The maximum intent approach simply selects a concept with maximum intent size from the PAR set. The larger the intent size, the larger the number of state/out-transitions shared by states in the extent of the selected concept. This heuristic choice maximises the number of symbol transitions to remove from each of the a DFA's relevant states (namely those in the concept's extent) states and replaces them with a failure transition at each state. However, it does not have to regard the number of states at which such a transformation will occur — i.e. it does not consider the size of the concept's extent. This approach is directly related to the hierarchical structure of the concept lattice's line diagram (the Hasse diagram), in that concepts with the largest intents typically have smallest extents and are therefore displayed in the lower parts of the diagram.

  To facilitate the implementation, the input PAR set can be stored in a list sorted in descending order of intent size.

- $c := MinExtentConcept(PAR)$;
  Like the previous criterion, this concept selection criterion is reflected in the hierarchical structure of a formal concept lattice's line diagram. Generally, concepts with the least number of objects (states) in their

extents tend to have more attributes (state/out-transitions) in their intents — i.e. small sets of objects tend to have more common properties than larger sets. Although this criterion does not result into an identical selection to $MaxIntentConcept(PAR)$, it also aims to prioritise transformations on sets of states where there are likely to be larger numbers of symbol transitions removed and replaced by a failure arc per state.

To facilitate the implementation, a list sorted in descending order of extent size is a suitable data structure for the PAR set.

- $c := MaxIntMaxExtConcept(PAR)$;
  This approach for symbol transition removal is based on refining the idea of maximum arc redundancy — also is a derived and constrained extension of the *MaxIntentConcept* criterion. It was proposed after noting that the *MaxIntentConcept* criterion often requires a selection from one of several concepts because their intents have the same size. Rather than selecting one of these arbitrarily, a concept with maximum extent is selected. At first, all concepts with tied maximum intent are provisionally selected, and from them the concept with the maximal extent is chosen. Clearly, this approach seeks a middle ground between the *MaxARConcept* and the *MaxIntentConcept* criteria. The motivation is that in addition to selecting concepts with the largest attribute size (for optimum transitions removed per state), we also seek to select concepts that can potentially allow the highest number states to become sources for a failure transition.

  For implementation purposes, the PAR set can be stored in two dimensional arrays. The first row contains an array with all concepts having the highest intent size. The second row holding an array of all concepts with the second highest intent size and so on. Furthermore, in each row array the concepts can be sorted in descending order based on the extent size.

3.2.2 Select a Target State t from Concept c

Once any one of the foregoing criteria has been applied, an arbitrary state in the extent of the selected concept c may, in theory, become the target state. Failure transitions from each of the remaining states in the extent to the selected target state will subsequently be installed provided that a divergent failure cycle is not produced. The criterion proposed here for selecting the target state of a failure transition t is to select a state in the extent closest to the DFA's start state. This simple criterion was chosen over more complicated possibilities for ease of implementation. In Algorithm 3.2 a line of code states the criterion in abstract form as:

$$t := ClosestToRoot(c);$$

When the *ClosestToRoot* criterion is invoked in DHA with the trie-embedded AC-opt DFA as input, t becomes the state in $ext(c)$ with the least unit depth[2]. However, with an arbitrary complete DFA as input to DHA, the notion of an embedded trie falls away and therefore also the notion of "depth of a state". Instead, the state in $ext(c)$ that has the shortest distance from the start state is selected as t, and the computation of this distance is slightly more complicated than computing the depth of a trie state. We therefore have the following two variants of the *ClosestToRoot* criterion for selecting the target state:

- $t := StateWithLeastDepth(c)$;
  The symbol transitions of an Aho-Corasick failure automaton form a structure made of a forward trie. At an arbitrary state, AC-fail FDFA's failure transitions are tailored to fail towards target states that have a depth less than the current state. In fact, the AC-fail algorithm allows a state to fail back to a state that holds the longest prefix that is equal to a suffix of the path that is traversed currently.

[2]The depth of a state in a trie is the total number of transitions on a path to that state that begins at the start state.

The "longest prefix of a suffix" property can be demonstrated as such. Given a string $w = t.u.v$, the AC-fail algorithm directs a failure transition from a state corresponding to the last character of word w, to another state that corresponds to another word of the form $(v.z)$ with longest prefix v that equals suffix v of w. The state is chosen so that the length of v is at a maximum.

While using the *StateWithLeastDepth* criterion, failure transitions are attached as follows. From the extent of the concept c, the state with the least depth is selected to be target state, t, of a failure transition. The remaining states become potential failure transition source states.

The difference between these two methods of selecting a target state of failure transition criteria is described in this paragraph. Assuming states p, q and r with $depth(p) < depth(q) < depth(r)$ to be states in ext(c) from a selected concept c's in PARset. The *StateWithLeastDepth* approach will select p to be the failure transition target state ahead of both q and r because p has the least depth. While, the Aho-Corasick algorithm will direct failure arcs from r to q (and not p) when we assumed that suffix to r is the longest prefix to q. Cases of correspondence between the AC-fail algorithm and the DHA criteria will be when the state p is the longest prefix corresponding to suffix r.

- $t := StateWithLeastDistanceFromStartState(c)$;
  Selecting a failure transition target state with minimum trie-depth is restricted trie-based structures such as the AC-opt DFAs. As it will be seen later, our final objective is not to be restricted with benchmarking DHA FDFAs against the AC-fail FDFAs, but also to the general case FDFA. As such, we select the failure transition target state by calculating the minimum/shortest distance from the start state. It should be noted that in the Aho-Corasick automata, the expected shorted distance from a root state to a state must be equal to the respective trie depth. Several known algorithms that calculate the shortest distance between two states in an arbitrary graph and three examples

include algorithms by Dijkstra [47], by Belman-Ford [48] and by Floyd-Warshall [49]. A complete DFA must have all valid states reachable, and because of that, in this research Dijkstra's shortest path algorithm is employed, as given in Drozdek [50, p. 394].

3.2.3 A Minor Modification

The new version of the algorithm will not delete any transition from the initial state q_s of the input DFA and this is because of the previously discussed modification: $t := ClosestToRoot(c)$. It was therefore deemed desirable to improve the efficiency of DHA by modifying the following line from the abstract DHA. The line $O := Q$ is modified to:

$$O := Q \backslash q_s;$$

Recall that the set O keeps track of the states that may still become the source of a failure transition. In the AC-fail algorithm, the start state is generally not assigned as the source of a failure transition. By excluding the start state q_s from being considered as a candidate to be the source of a failure transition in DHA, it was hoped that the DHA would produce FDFAs that are closer in structure to the AC-fail algorithm's FDFAs.

3.3 Summary

In this chapter, various modifications to adapt the DFA-homomorphic algorithm to effectivelly and deterministically remove symbol transitions were presented. The effected changes on DHA affect only the selection of concepts from the input PAR set and the choosing of a failure transition target state. A toolkit that implements DHA algorithms is available online[3]. The following chapter describes the approaches of testing of currently discussed DHA

[3] http://madodaspace.blogspot.co.za/2016/02/a-toolkit-for-failure-deterministic.html

FDFA variants against the well-known optimal symbol transition reducer called AC-fail FDFA and against a generalised random FDFA.

Chapter 4

Methods and Tools Used

In this chapter we present an exploratory research approach to compare the Aho-Corasick (AC) failure machine against the DHA FDFA variants (described in the previous chapter) and D^2FA. As stated by Olivier [51], in exploratory research, an experiment is designed, conducted and a theory may be proposed to explain observations.

In the section below, the dissertation provides a number of new formal propositions, proofs and properties that are relevant to the AC based experiments. Subsequent sections present a detailed description of the quantitative assessment approaches that are employed in the experiments. Additionally, the software and hardware tools used in this research are summarised and critiqued.

Much of the content covered in this chapter is tailored towards comparing AC automata against DHA FDFA but some content is general and covers details relating to the general case FDFA tests as well. Specific aspects of the general case FDFA assessments that are not covered here will be looked at in Chapter 6.

4.1 AC-fail FDFA Arc Reduction Characteristics

Formal propositions, proofs and properties are presented here that relate to transition reduction of AC-fail FDFAs in comparison to their AC-opt DFA counterparts. These AC preliminaries govern expectations of the empirical results when comparing the DHA automata against the AC automata. These properties and their corollaries were derived specifically as part of this research.

Proposition 4.1.1. *If Q, δ and $\mathfrak{f}$ are the set of states, symbol transition function and failure transition function respectively of an FDFA generated by the AC-fail algorithm, then:*

$$|\delta| \leq (|\Sigma| + (|Q| - 1)) \tag{4.1}$$

and

$$(|\delta| + |\mathfrak{f}|) \leq (|\Sigma| + 2(|Q| - 1)) \tag{4.2}$$

Proof. Let ℓ_1 be the number of looping symbol transitions at the start state q_s for an AC-fail FDFA (i.e. the count of transitions such that $\delta(q_s, a) = q_s,\ a \in \Sigma$). Let $\ell_{>1}$ be the number of remaining symbol transitions in AC-fail FDFA. The $\ell_{>1}$ symbol transitions form a trie. Therefore, for each state (excluding the start state) there is exactly one inbound symbol transition. Thus, easily obtaining $(|Q| - 1)$ transitions, that is, $\ell_{>1} = (|Q| - 1)$. The root state has $\leq |\Sigma|$ looping symbol transitions i.e. $\ell_1 \leq |\Sigma|$. The remaining start state symbol transitions (which are of the form $\delta(q_s, a) \neq q_s$) are associated with the $\ell_{>1}$ transitions — thus they have been counted from the first argument. Hence AC-fail FDFA's symbol transitions size is;

$$\begin{aligned} |\delta| &= \ell_1 + \ell_{>1} \\ &= \ell_1 + (|Q| - 1) \\ &\leq |\Sigma| + (|Q| - 1) \end{aligned}$$

An AC-fail FDFA has $(|Q|-1)$ valid failure transitions, which are failure transitions for all states excluding the initial state. The initial state cannot have a failure transition because it is a root node of the trie so there is no keyword P that can be formed by empty string ϵ. Finally, the total number of AC-fail FDFA transitions is;

$$\begin{aligned}(|\delta|+|\mathfrak{f}|) &= \ell_1 + 2\ell_{>1} \\ &= \ell_1 + 2(|Q|-1) \\ &\leq |\Sigma| + 2(|Q|-1)\end{aligned}$$

□

Corollary 4.1.2. *Without loss of generality, for AC-fail FDFA with a large* $|Q|$,

$$|\delta| \approx |Q| \tag{4.3}$$

and

$$(|\delta|+|\mathfrak{f}|) \approx 2|Q|. \tag{4.4}$$

The complete DFA generated from the AC-opt DFA has the same number of states as the AC-fail FDFA, it has $|\Sigma| \times |Q|$ symbol transitions. For large $|Q|$, the proportion of symbol transitions saved by the AC-fail FDFA over the symbol transitions of the AC-opt DFA is thus bounded from above by *Property 4.1.3*. Similarly, the ratio of AC-fail FDFA symbol transitions (the trie) over those of the AC-opt DFA is bounded from above by *Property 4.1.4*.

Property 4.1.3. *The ratio* $R_{(|\delta|+|\mathfrak{f}|)}$ *of the overall transition contained by AC-fail FDFA over AC-opt DFA transition size is approximated by;*

$$R_{(|\delta|+|\mathfrak{f}|)} \approx \frac{(|\Sigma|-2)}{|\Sigma|}$$

This can be easily be derived from Equation 4.4 and the AC-opt DFA size as

follows.

$$\begin{aligned} R_{(|\delta|+|\mathfrak{f}|)} &= \frac{(AC\text{-}opt\_\,|\delta|) - (AC\text{-}fail\_\,(|\delta| + |\mathfrak{f}|))}{(AC\text{-}opt\_\,|\delta|)} \\ &\approx \frac{(|\Sigma| \times |Q|) - 2|Q|}{|\Sigma| \times |Q|} \\ &= \frac{(|\Sigma| - 2) \times |Q|}{|\Sigma| \times |Q|} \\ &= \frac{(|\Sigma| - 2)}{|\Sigma|} \end{aligned}$$

Property 4.1.4. *The proportion $R_{(|\delta|)}$ of AC-fail FDFA symbol transition size over AC-opt DFA symbol transitions is estimated by;*

$$R_{(|\delta|)} \approx \frac{(|\Sigma| - 1)}{|\Sigma|}$$

Similarly, this can be derived as shown below.

$$\begin{aligned} R_{(|\delta|)} &= \frac{(AC\text{-}opt\_\,|\delta|) - (AC\text{-}fail\_\,|\delta|)}{(AC\text{-}opt\_\,|\delta|)} \\ &\approx \frac{(|\Sigma| \times |Q|) - |Q|}{|\Sigma| \times |Q|} \\ &= \frac{(|\Sigma| - 1) \times |Q|}{|\Sigma| \times |Q|} \\ &= \frac{(|\Sigma| - 1)}{|\Sigma|} \end{aligned}$$

4.2 Comparing the Aho-Corasick Automata with DHA FDFA

The workflow for the experiment is depicted in Figure 4.1. Firstly, a keyword set, P, is created characterised by some controlling constraints which are detailed in the next subsection. Upon inputting the keyword set into

SPARE Parts toolkit developed by [52, 53], two Aho-Corasick automata are produced — an AC-opt DFA and a language-equivalent AC-fail FDFA, respectively. The AC-opt DFA serves as input into any one of the DHA variants and/or as input into an implementation of the D^2FA algorithm. Additionally, the AC-opt DFA is inputted into FCART (developed by Buzmakov and Neznanov [54]) in a converted form as a state/out-transition context. Then FCART produces concepts which are used as input into DHA. The resulting FDFAs are DHA-FDFA (any variant) and D^2FA, respectively. To convert an AC-opt automaton into an FDFA, DHA implements the *StateWithLeastDepth* method for *ClosestToRoot* heuristic. Meanwhile, AC-fail FDFA becomes the experimental control FDFA, serving as the optimal standard. The DHA FDFAs and D^2FA are compared against it.

The failure transitions and symbol transitions of the AC-fail FDFA and the other FDFAs are compared against one another. The comparison is both in terms of the extent to which transitions match one another and in terms of the overall number of transitions.

4.2.1 Building an AC-fail FDFA out of a Keyword Set

From a set of keywords (also referred to as a pattern set), P, we use the AC-opt algorithm in the SPARE-Parts toolkit to construct an AC-fail FDFA that recognises the language Σ^*P. Similarly, we use the AC-fail algorithm in the SPARE-Parts toolkit to construct an optimal AC-opt DFA that recognises the same language.

It can easily be demonstrated that if there are no overlaps between proper prefixes and proper suffixes of keywords in a keyword set, then the failure transitions of the associated AC-fail FDFA will all loop back to its start state. This is because such a keyword set will cause the AC-fail algorithm to construct a trie that fans out from the start state with a branch for every keyword in the keyword set. The *ClosestToRoot / StateWithLeastDepth* heuristic implemented in DHA will result in FDFAs with similar properties.

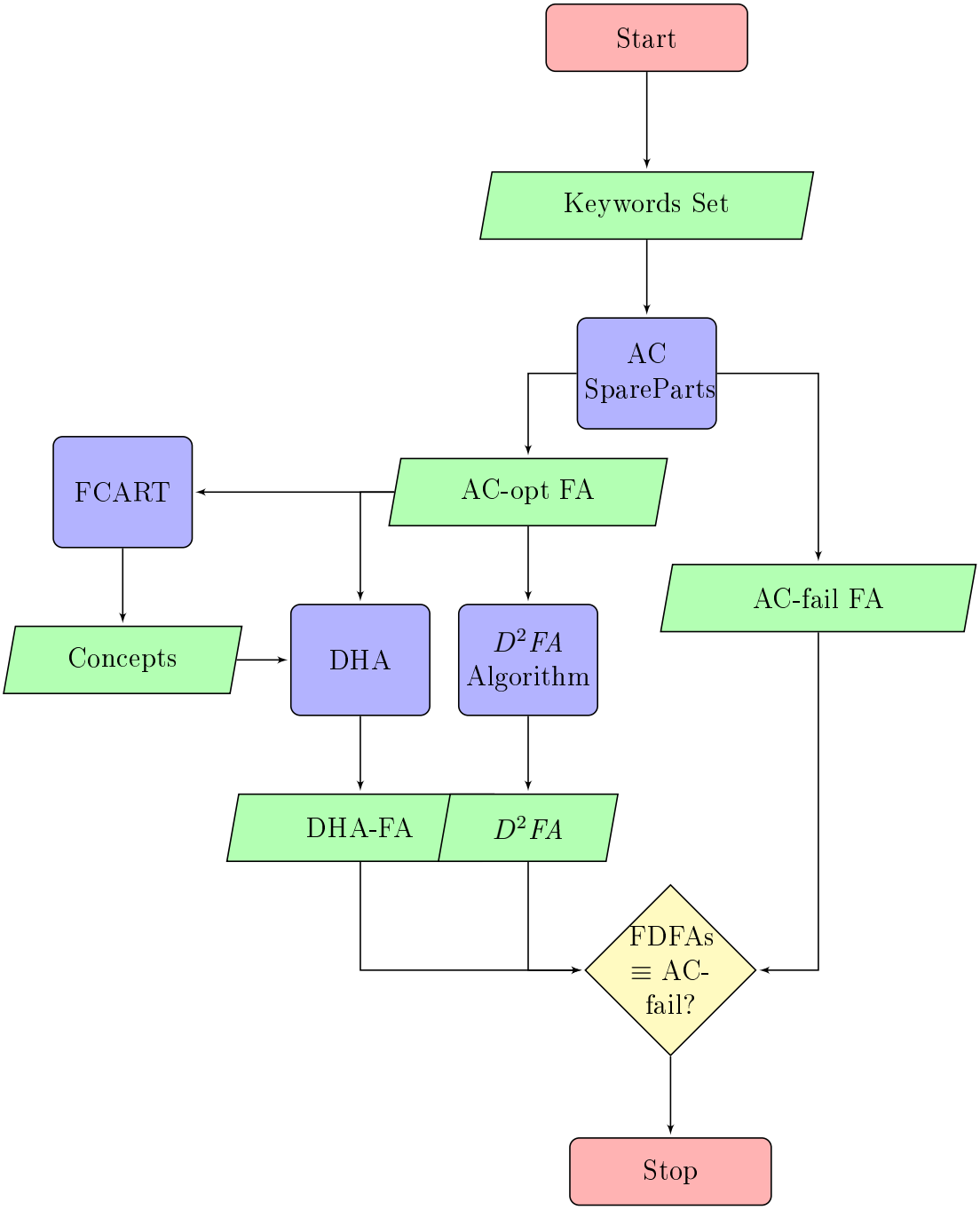

Figure 4.1: Comparing DHA-FDFAs and D^2FA against AC-fail FDFA.

To avoid keyword sets that lead to such trivial AC-fail FDFAs, it is important to create keyword sets that ensure variation in the failure transition destinations of the AC-fail FDFAs that are generated.

Such keyword sets must cause the AC-fail algorithm to construct tries which have several states containing more than one symbol out-transitions — not only to limit this property to the start state only. The keywords must also force the AC-fail algorithm to enforce the back-tracking failure transitions to fail towards states other than the initial state. These AC-fail FDFA characteristics are listed in the next paragraphs.

Given a pattern set P, and strings $s_i, s_j \in P$ with $s_i = p_i.q_i.r_i$ and $s_j = p_j.q_j.r_j$. Accordingly, the p, q and r terms indicate strings that are prefixes, substrings and suffixes. Alternatives for adding a new keyword into the AC-fail FDFA trie are listed below. Here, s_i is some keyword already belonging to the acyclic AC-fail FDFA trie and s_j is the new keyword to add into the trie.

1. If s_i is a proper prefix of s_j i.e. $p_j = s_i = p_i.q_i.r_i$, then from the final state of $s_i = p_j$ extend the path by $q_j.r_j$.

2. if s_j is a proper prefix of s_i i.e. $p_i = s_j = p_j.q_j.r_j$, then change the last state of the path labelled by p_i to a final state.

3. If s_i is a substring of s_j with $p_i = p_j$, at the last state of p_i path then create a new branch out of s_i extending p_i to $q_j.r_j$.

4. Otherwise add s_j as a completely new path from the initial state.

The 'suffix of a prefix' property for building AC-fail FDFA's failure arcs is briefly described. Let $s_k = p_k.q_k.r_k$ to be a string which is recognized by a path in a trie. In order to orient a failure arc from the state that recognizes s_k, the AC-fail algorithm finds some prefix p_i from the trie such that $r_k = \textit{longest}\ p_i$. (Recall that r_k is a suffix of s_k.) Then a failure transition is directed from the last state recognizing s_k to last state of the prefix p_i. Note also that the *longest* p_i may be p_k — a prefix of s_k.

Further details for creating an Aho-Corasick machines are available from [13, 23].

4.2.2 Measurements of Matching Transitions and Transition Reduction

Quantitative measurements of transition reductions entail measuring the number of transitions saved by the FDFAs produced by the various algorithms relative to the AC-opt DFA from which they were derived. On this account, this research aims to measure the following: the number of symbol transitions saved, and the overall transition reduction after introducing failure transitions. As a measure of the overall number of transitions saved by the FDFA in comparison to the DFA, the ratio of FDFA transition size over the DFA transition size (i.e. $FDFA(|\mathfrak{f}|+|\delta|)$ per $DFA(|\delta|)$) is calculated.

As indicated by Corollary 4.1.2 and Property 4.1.3 an AC-fail FDFA contains $2|Q|$ transitions and the ratio of AC-fail FDFA transitions to AC-opt DFA transitions is bound from below by the expression; $(|\Sigma|-2)/|\Sigma|$. These expressions, too, should serve as useful indicators of whether the DHA and D^2FA generated FDFAs have been efficiently produced. For ease of reference, results will be given as percentages.

After examining results obtained from Aho-Corasick automata, the aforementioned measurements will be applied to FDFAs produced from general DFAs. As will be described in Chapter 6 these general DFAs will not be entirely randomly selected, since there will typically be limited opportunities in such cases for replacing symbol transitions by failure transitions. Instead, special efforts will be made to select DFAs that are random in character but nevertheless also offer many opportunities for generating failure transitions.

Additional quantitative measurement will be conducted in order to determine the correspondence between the DHA generated FDFAs in comparison to AC-fail FDFAs. This is done by finding the extent to which symbol tran-

sitions and failure transitions *match* between the DHA generated FDFAs and the AC-fail FDFAs. Notice that it is possible for two different FDFAs to have the same set of states, the same alphabet set, the same number of failure transitions and to recognise the same language, but to locate these transitions differently between states. This difference in FDFA transitions correspondance is depicted as is, because expressing percentage ratios conceals the extent of transition mismatches.

The empirical details for this are shown and discussed in the next chapter. Note that the D^2FA FDFAs will also be included in the comparative study.

4.3 The Experimental Environment

The choice of software and hardware tools affect experiments in several ways. They are potential determinants of the strengths and weaknesses of the research study since they may enforce certain conditions on the experiments. In computing experiments, software and hardware tools used will be restricted by the financial costs for the study and may as a result be constrained and limited in terms of the input data and the cost of execution times of the programs. Table 4.1 lists all major tools used in this present study. This section provides brief descriptions of the following softwares: SPARE Parts, FCART and *R*.

Table 4.1: The Environment for Experimentation

Tool	**Specification details**
Computer Specifications	*Intel i5; Dual Core CPU machine*
Operating System	*Linux Ubuntu version 14.4; 64 bits*
Programming language	*C/C++*
Compiler	*GCC version 4.8.2*
Automata Generator	*SPARE Parts*
Concept Lattice generator	*FCART version 0.9*
Data Analyser	*R version 3.1.1*

SPARE parts is a software toolkit that provides an implementation of keyword pattern matching algorithms. (Refer to [19, 52, 53] for more details.) The toolkit provides implementation of finite pattern set matching algorithms such as Aho-Corasick, Commentz-Walter, Knutt-Morris-Pratt, and factor oracles. The first version of SPARE parts was implemented in the C++ programming language in 1995 by Bruce Watson. Later, in 2003 together with Loek Cleophas (refer to [19, Chapter 5]), they extended the toolkit and coined a new name, SPARE time. The latter version was written in more modern C++. The toolkit is available online[1].

The software system used to build formal concept lattices is called Formal Concept Analysis Research Toolbox (FCART). Refer to [55, 54, 56, 57] for more details about FCART. FCART was developed at the National Research University Higher School of Economics (Russia) and the project is led by Alexey Neznanov. FCART is an integrated environment that can be applied in many areas such as data mining and knowledge discovery. It has simple means for importing and exporting of data and for data pre-processing. This toolkit was developed within the Microsoft and Embarcadero programming environments and different programming languages such as C++, C#, Delphi, Python were used. For scripting purposes the developers used languages such as Delphi Web Script and Python.

FCART was given preference because of several reasons that are discussed below. FCART is able to display on screen the resulting concept lattice line (Hasse) diagram, which is advantageous for visualizing the concept lattice structure. Like many FCA tools such as Concept Explorer, FCART accepts contexts as Comma Separated Value (csv) files as a simple binary relational table between objects and attributes. Moreover, FCART produces the state/out-transition concepts, which are input to DHA, in an XML file. The XML format can be parsed by several XML parsers suitable for different programming languages — for example the *xcerce* XML parser library for C++.

[1]SPARE Parts can be found at: *http://fastar.org/main.php?button=spare_time*

R is a very useful quantitative data analysis tool. R is an open source project with many contributors (collaborated by the R Core Team [58]) that was initially developed by Robert Gentleman and Ross Ihaka. R can present visualizations of data in several formats — for example in the form of tables, bar charts, line graphs, etc. Moreover, R effectively and efficiently provides statistical summaries of data, for example calculating central tendencies of data such as mean values, median. A very good feature of R is producing boxplots from a set of data. In a boxplot, the median, range, quartiles and 'outliers' are visually represented. As a result, one obtains a clear visual indication of the data's central tendencies, its distribution and the extreme cases in the data. Such a summary of the data was found to be very instructive. Hence, by using boxplot, complex empirical data can sometimes be interpreted in interesting ways. Throughout this dissertation, empirical results are generally presented as graphs produced using R. More details about R and its usage can be found online[2] and from publications such as [59, 58].

4.4 The Experimental Data

In the AC experiments of this dissertation, the primary input data are keyword sets. These produce as output data, DFAs and the FDFAs. Since the DFAs are important secondary data sources, their construction is well-controlled by constructing the keyword input data as discussed in Section 4.2.1. We also take into account the need to generate a statistically justifiable number of (F)DFAs for the experiments.

To construct the keyword sets, alphabet sizes of four ($\Sigma = 4$) and ten ($\Sigma = 10$) characters respectively were used. This selection ensured that the resulting state/out-transition concept lattices could be generated with the available hardware and software[3]. Further explanations about this factor is

[2] R is currently available at: *https://www.r-project.org/about.html*

[3] It was empirically determined that with an alphabet size of about 15 the available software was stretched to its limits.

given in the next section.

To construct a keyword set of size N, an initial N random strings are generated[4]. Each such keyword has random characters taken from the alphabet and random length is within the range of a minimum of 5 characters and a maximum of 60 characters. However, for reasons given below, only a limited number of these N strings, say M, are directly inserted into the keyword set. Each keyword in the set is then incrementally grown to the desired size, N, crafted by setting the keyword to have any of properties of AC-fail FDFA listed in Section 4.2.1. The aforementioned properties are summarized as follows:

> Select a prefix of random length, say p, from a randomly selected string in the current keyword set. Create a random string, say w, from the set of strings not yet in the keyword set. Insert either $p.w$ or $w.p$ into the keyword set.

The string concatenation $p.w$ is generating a keyword that branches out from prefix a of another, already created keyword. The prefix p may be an existing keyword, and thus p is extended by w. The other concatenation $w.p$ is for creating failure transitions such that the "suffix of a prefix" property is met. Steps are taken to ensure that there is a reasonable representation of each of these three differently constructed keywords in a given keyword set. After obtaining these keywords, the SPARE Parts toolkit is used to create the AC-fail automata and the AC-opt automata.

The Aho-Corasick automata are constructed from the pseudo-randomly generated pattern set. The number of states for each (F)DFA is determined by the nature of the associated Aho-Corasick trie. As a result, the number of transitions cannot be defined before the Aho-Corasick automata (AC-fail FDFA and AC-opt DFA) are fully constructed — i.e. the number of states and transitions of the various (F)DFAs generated cannot be directly inferred from the size of the keyword set. Nevertheless, in this study, we characterise

[4]Note that all random selections mentioned use the C++ pseudo-random number generator.

the (F)DFA under test by the number of strings in its associated pattern set, P, since the size of AC-opt DFA is not known *a priori*.

Thus, the pattern sets generating AC-fail automata and AC-opt automata are produced in increments of five strings per pattern set — i.e. pattern sets sizes generated are 5, 10, 15, ..., 100 for $|\Sigma| = 10$. For each pattern set size, twelve Aho-Corasick automata are produced. Thus, for pattern sets of size five, twelve unique automata are produced, and each is created from five unique words. By doing so, the intention is to gain an idea of the behaviour of the automata at a given pattern set size. For example, an analysis of data can be enhanced by obtaining the mean values per pattern set size $|P|$ or by observing the distribution of the results per $|P|$. Consequently, the number of generated Aho-Corasick automata becomes 20×12 for each alphabet size, i.e. $|\Sigma| = 4$ and $|\Sigma| = 10$. In the subsequent chapter we delve deeper into the analysis of the data discussed in this section.

4.5 Constraints with Input Data

It is known that the trie of an AC-opt DFA has one inbound transition per state. Therefore, during the construction of a state/out-transition context table from such a DFA, it is most likely to produce many columns in the context for new attributes all derived from a single state. This in turn makes the states (objects) to have minor differences with each other — that is, many states have common attributes. This results into generating a very larger concept lattice. Consequently, this constrained our experimental exploration to even larger FDFA sizes.

Three concept lattice tools were tried — namely: FCART software, ConExp software and an implementation of the AddIntent algorithm. They were tried in order to determine a toolkit that can produce the largest number of concept lattices. They all have a limitation of performing poorly when generating concepts and concept lattices for AC-opt DFA with $|\Sigma| \leq 15$. This limitation encountered is common challenge of all concept lattice con-

struction algorithms. The concept lattice of a context can, in the worst case, grow exponentially in size — exponential, that is, relative to the number of attributes and objects. A further contributing factor for this disadvantage is computer hardware — for example small amounts of RAM and slow processors. Recently in Neznanov and Parinov [60], FCART have been tested to work in a distributed environment and with the availability of distributed computers this version of FCART can be helpful in processing big data sets. Due to time constraints and limited resources we did not use this approach to generate concepts.

4.6 Conclusion

This chapter described the techniques that are applied in this investigation. Relevant theories and their proofs were presented, the methods of comparing the failure automata were discussed as well as the tools used. Notably, the size of the data is constrained by the hardware as well as the formal concept software tools at our disposal. The next couple of chapters cover the empirical results for the experimentation described in this chapter. The next chapter compares the DHA and D^2FA algorithm against the Aho-Corasick automata types. The current chapter provided a foundation of the experimental results presented in the subsequent chapter.

Chapter 5

Transition Reduction for AC-opt DFAs

5.1 Introduction

Empirical results that were obtained by comparing the FDFAs from DHA variants with two Aho-Corasick automata types (AC-fail and AC-opt) are presented. The introduction provides a brief summary of the experiment.

Four types of FDFAs are produced from an AC-opt DFA using DHA. The DHA FDFAs characterised, respectively as the *MaxArcRedundancy (in short, MaxAR) FDFAs*, the *MaxIntent FDFAs*, the *MinExtent FDFAs* and the *MaxIntent-MaxExtent FDFAs*. The properties of these FDFAs are then further examined. We also include in this investigation transition data of the D^2FA FDFAs and also of the experiment's control FDFAs that were derived from the AC-fail algorithm.

For each experiment run, an AC-opt DFA and an AC-fail FDFA were derived using the SPARE Parts toolkit. Each pair of AC automata was recognising some keyword (pattern) set. The keyword sets were constructed with reference to properties of AC-fail FDFAs which are listed in Section 4.2.1. Details of the steps followed to construct the keyword set were given

in Section 4.4. It was pointed out that the maximum length of each keyword is 60 characters and the minimum length is 5 characters. The sizes of keyword sets are grouped in orders of five — that is, the number of strings per experiment is $[5, 10, 15, \ldots, 100]$. Moreover, experiments were carried out with two different alphabet sizes, these sizes being either $|\Sigma| = 10$ or $|\Sigma| = 4$. The results are primarily referenced in the text below.

For each of the produced DHA and D^2FA FDFAs, transitions are compared against both types of AC automata.

- In the first tests, transition reductions are investigated. They cover the following:
  - symbol transitions removed from the AC-opt DFAs by the FDFA algorithms;
  - failure function size differences between the AC-fail FDFA and the various FDFAs; and
  - the sum of FDFA failure and symbol transitions (the overall FDFA transitions) compared against the total number of DFA transitions.

  In the latter case, the overall transition reduction is assessed when using an FDFA instead of a language-equivalent DFA.
- The second investigation is about measuring differences between the transitions of AC-fail FDFAs and other FDFAs. For each keyword set, the extent of transition equivalence between the AC-fail FDFA and the other language-equivalent FDFAs is identified. In this case, the focus is on the equivalence of the *placement* of the transitions. This means that an AC-fail FDFA and some other FDFA are equivalent in terms of the symbol transition on $a \in \Sigma$ at state $q \in Q$ if and only if $\delta(q, a)$ of the AC-fail FDFA is equal to $\delta(q, a)$ of the other FDFA. Similarly, failure transitions are equivalent if and only if AC-fail FDFA's $\mathfrak{f}(q)$ is equal to the other FDFA's $\mathfrak{f}(q)$.

For further details about the nature of the experiments, refer to Chapter 3 and Chapter 4. Chapter 3 presented descriptive details of the DHA, indicating how the four FDFA variants are generated. In Chapter 4 we provided the background and the constraints of this experiment.

5.2 The Results

The transition comparisons outlined in the previous section are displayed in the graphs in this section. Moreover, the graphs are discussed. In certain cases the data of the various DHA FDFAs are practically the same. In such cases we will use the data from one of these DHA FDFAs to represent the data of all of remaining FDFAs in the graphs. This is will be common occurrence in respect of the following three DHA FDFA heuristics: MaxIntent, MinExtent and MaxInt-MaxExt. The presented results are for experimental runs in respect of automata for which $|\Sigma| = 10$ and $|\Sigma| = 4$. Detailed examination of the results will be provided in the next section.

5.2.1 Differences in Transition Sizes

This section is about presenting measurements of the transitions deleted from AC-opt DFAs by the FDFA generating algorithms under investigation. The number of symbol transitions removed by each of the different FDFA generating algorithms is counted, the differences in number of failure transitions between the AC-fail FDFAs and other FDFAs are measured, and lastly, a comparison of FDFA sizes (as measured by the total number of transitions) and AC-opt DFA sizes is given.

Symbol Transitions Removed

Since using an FDFA for string recognition potentially reduces computer memory required to store the automaton, the more symbol transitions re-

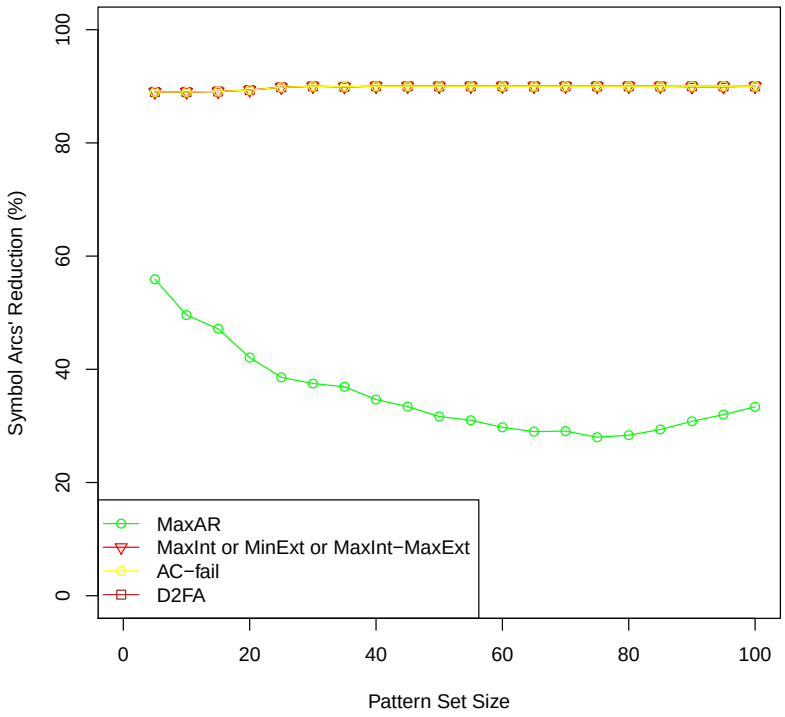

Figure 5.1: The *average* FDFA reduction in *symbol* transitions as a percentage of AC-opt DFA transition sizes ($|\Sigma| = 10$)

moved from a DFA by an FDFA-generating algorithm, the better the algorithm can be said to have performed. This investigation seeks to determine the extent to which the general DHA and D^2FA algorithms remove symbol transitions from AC-opt DFAs to produce FDFAs. The use of AC-opt DFAs for this exercise, as opposed to randomly produced DFAs, is particularly useful because it is known *a priori* that of all the possible FDFAs that are language-equivalent to an AC-opt DFA for a given keyword set, the AC-fail FDFA for the keyword set has the fewest possible symbol transitions.

Figure 5.1 shows the average percentage of symbol transitions removed from the AC-opt DFAs by the various FDFAs for various keyword set sizes that are based on an alphabet size $|\Sigma| = 10$. These FDFA variants are the AC-fail FDFAs, the D^2FA FDFAs and all the DHA FDFA variants. For each

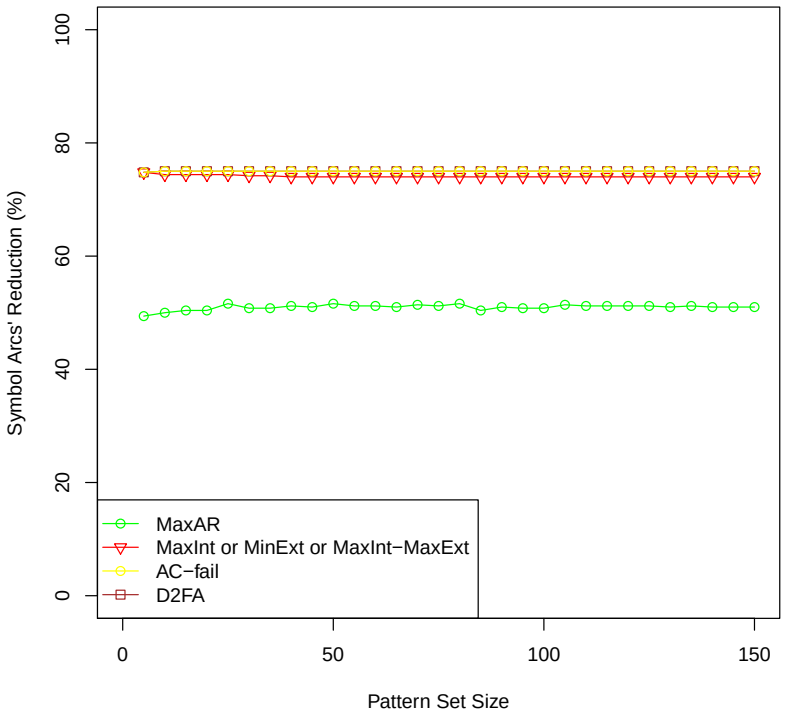

Figure 5.2: The *average* FDFA reduction in *symbol* transitions as a percentage of AC-opt DFA transition sizes ($|\Sigma| = 4$)

pattern set size in $[5, 10, \ldots 100]$, the average is computed over the 12 data samples generated for that pattern set size.

Like the benchmark AC-fail FDFAs, D^2FA FDFAs and three of the DHA FDFA variants (the MaxIntent FDFAs, MinExtent FDFAs and MaxExtent-MaxIntent FDFAs) show that the percentage of symbol transitions are removed from AC-opt DFAs is more or less a constant at between 89% and 90%.

Figure 5.2 gives the corresponding graphs when $|\Sigma| = 4$. In this case, the symbol transitions removed constantly stand at approximately three-quarters for the FDFA types which were discussed in previous paragraphs. Note that the DHA FDFAs have a slightly lower average of 74% symbol transitions

removed.

The observations from the two previous paragraphs corroborate Property 4.1.4. According to Property 4.1.4 the estimated percentage of symbol transitions that are removed by AC-fail FDFAs compared to AC-opt DFAs is $\frac{(|\Sigma|-1)}{|\Sigma|} \times 100$. The other aforementioned FDFAs also match this property very well.

The MaxAR DHA FDFAs did not efficiently reduce the number of AC-opt DFA symbol transitions, both for $|\Sigma| = 10$ and for $|\Sigma| = 4$. It was by far no match for the other FDFAs under discussion. When $|\Sigma| = 10$, the MaxAR DHA FDFAs removed an estimated ceiling of 60% of the DFA symbol transitions, declining to a floor of 20% symbol transition reductions. When $|\Sigma| = 4$, the MaxAR DHA FDFA symbol transitions reductions remain very low.

"Useless" Failure Transitions

Suppose that a state $q \in Q$ of an AC-opt DFA is such that none of its out-transitions qualify for replacement with a failure transition in a language-equivalent FDFA. This would be the case if the out transition from q for every $a \in |\Sigma|$ has a target state that differs from the target state of every other out-transition on that same symbol a from other states in the DFA. Under such circumstances, the AC-fail algorithm as implemented by the SPARE Parts toolkit nevertheless inserts a failure transition from q to the start state — i.e. all AC-fail FDFA states have failure transitions. The initial state, q_s, which has $|\Sigma|$ transitions is an exception from having a failure transition. The failure function of every AC-fail FDFA is therefore of size $|\mathfrak{f}| = (|Q|-1)$. But some of the failure transitions, though valid, may be "useless" since a failure transition is inserted at a state, even when there is no valid symbol transition to be considered[1].

[1] "Useless" failure transitions do not, of course, mean the finite state machine is nondeterministic.

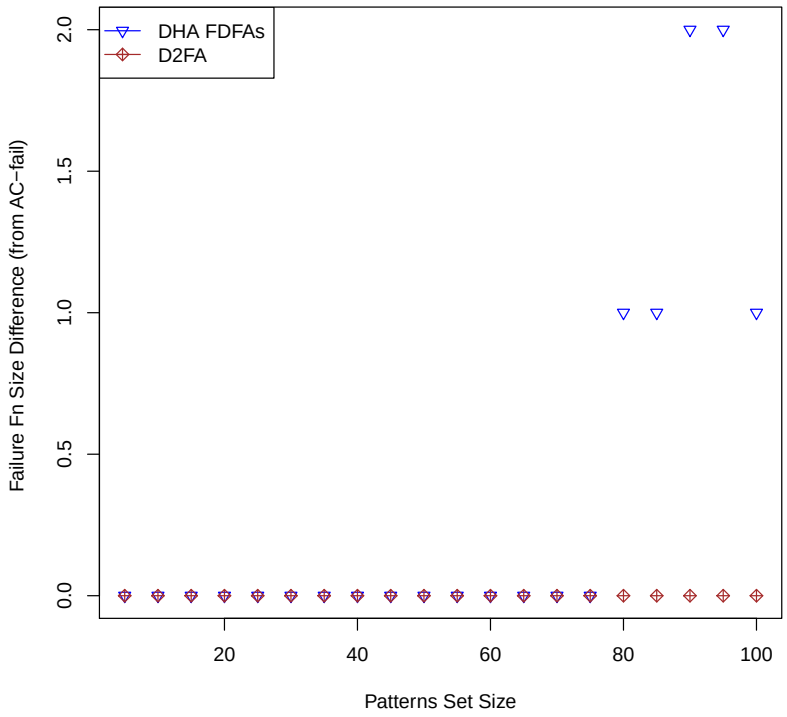

Figure 5.3: The *maximum difference* in *failure* function sizes of DHA FDFAs and D^2FAs from AC-fail FDFAs when $|\Sigma| = 10$.

These so-called "useless" failure transitions are not generated by DHA. DHA does not introduce a failure transition without removing at least two symbol transitions. On the other hand the D^2FA algorithm, like the AC-fail algorithm, also allows for the generation of "useless" failure arcs.

The extent of these "useless" failure transitions is depicted in Figure 5.3 for $|\Sigma| = 10$ data sets and in Figure 5.4 for $|\Sigma| = 4$ data sets, respectively. The figures show cases of maximum differences in failure function sizes of AC-fail FDFAs and DHA FDFAs per pattern set.

Figure 5.3 depicts the following for $|\Sigma| = 10$. For most pattern set sizes, AC-fail FDFAs did not have any useless failure transitions. There are, however, several exceptions. Not a single one of the twelve AC-fail FDFAs of the

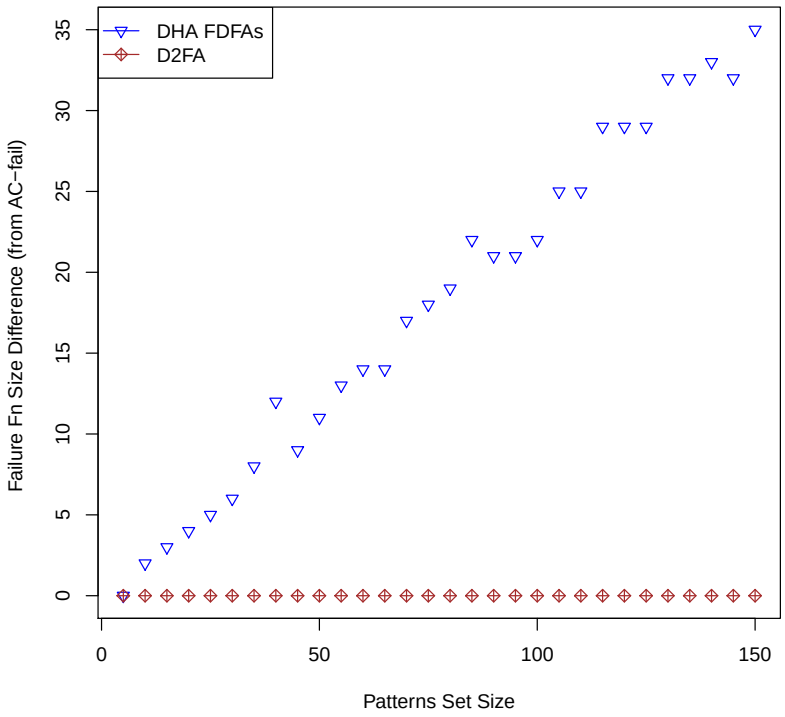

Figure 5.4: The *maximum difference* in *failure* function sizes of DHA FDFAs and D^2FAs from AC-fail FDFAs when $|\Sigma| = 4$.

data sets with pattern set sizes between $5 - 50$ contained a useless failure transition. In the case of data sets with pattern set sizes of 55, 80, 85 and 100, at least one instance of an FDFA that contained a useless failure transition was encountered. Furthermore, Figure 5.3 shows that in some instances, two (but no more than two) useless failure transitions were generated when using SPARE Parts to construct AC-fail FDFAs for pattern set sizes 90 and 95 respectively. Since we were recording the maximum number of useless failure transitions, we note in passing that there may be other AC-fail FDFA with the pattern set sizes of 90 or 95 that have a single useless failure transition. Furthermore, none of the pattern sets resulted in AC-fail FDFAs with more than two useless failure transitions.

Referring now to Figure 5.4, it depicts the following for $|\Sigma| = 4$ data

sets. In general, the graph shows that the maximum number of useless failure transitions that AC-fail FDFA possessed can be estimated to be linearly dependent on the pattern set size. The data set recorded a maximum difference of 35 failure transitions for data with pattern set size of 150. The data demonstrate that for smaller alphabet the chances of useless transitions produced by the AC-fail algorithm are higher.

Overall Transition Reduction: Symbol and Failure Transitions

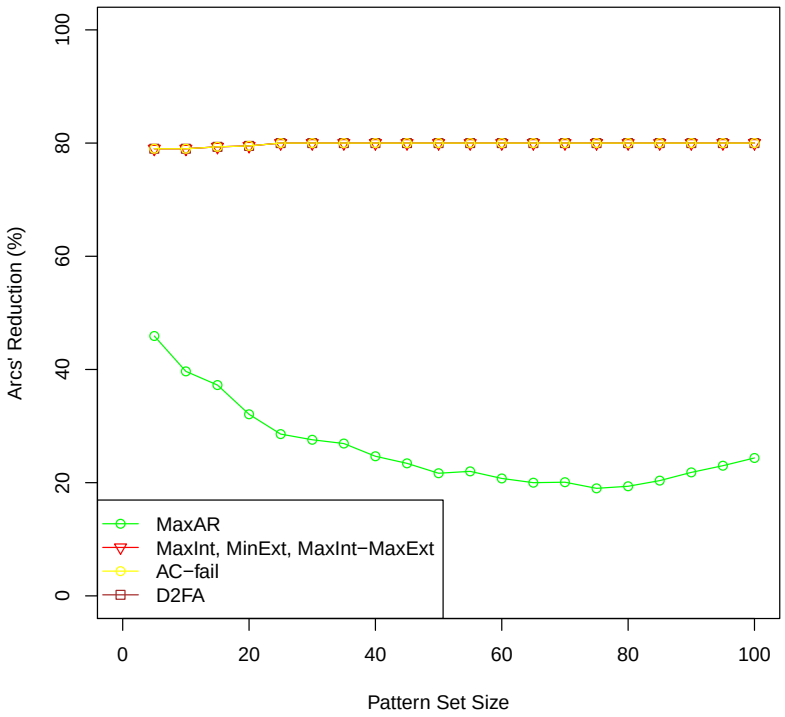

Figure 5.5: The *average* FDFA reduction in *total* number of transitions as a percentage of AC-opt DFA transition sizes ($|\Sigma| = 10$)

Figures 5.5 and 5.6 depict, for alphabet sizes 10 and 4 respectively, the overall number of transitions by which the FDFAs under test reduce the total number of AC-opt DFA transitions. By "overall number of transitions" in an

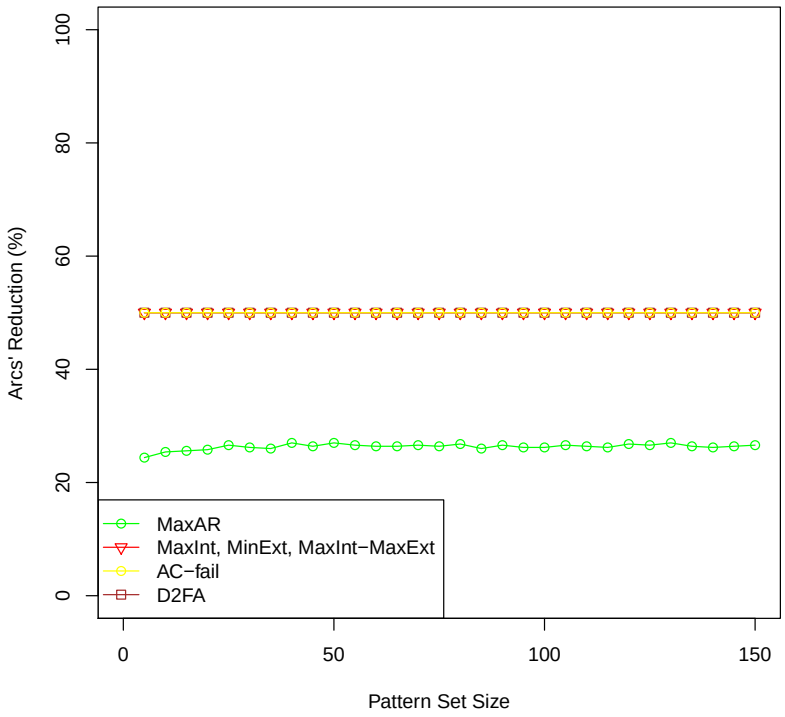

Figure 5.6: The *average* FDFA reduction in *total* number of transitions as a percentage of AC-opt DFA transition sizes ($|\Sigma| = 4$)

FDFA we mean sum of its symbol transitions and failure transitions.

As demonstrated in equation 4.4 of Corollary 4.1.2, when $|Q|$ is large then the total number transitions of AC-fail FDFAs is approximately given by $2|Q|$. Thus, the expectation is an addition of $|Q|$ failure transitions to AC-fail FDFA's $|Q|$ symbol transitions found in first subsection of Section 5.2.1. As is suggested by Proposition 4.1.3, this is expected to lead to a reduction of AC-opt transitions to FDFA transitions by approximately $\frac{|\Sigma|-2}{|\Sigma|} \times 100\% = 80\%$ when $|\Sigma| = 10$ and approximately 50% when $|\Sigma| = 4$.

With $|\Sigma| = 10$ data sets, the minimal AC-fail FDFAs indeed attain this predicated average transition reduction of about 80% over all sampled pattern set sizes. The DHA FDFA variants (except for MaxAR FDFA) and D^2FA

FDFA generally track this performance identically for FDFAs with $|\Sigma| = 10$.

By way of contrast, at $|\Sigma| = 10$ data sets, the MaxAR heuristic produces FDFAs barely achieve a 50% average reduction for small keyword set sizes. This reduction declines with increasing keyword set size to below 20% for a sample size of about 75, after which is there is some evidence that it might improve slightly.

When we consider FDFAs with an alphabet size of 4, the overall transition reduction conforms to the same theoretical behaviour, becoming approximately 50% average reduction for all failure automata. Again, the MaxAR FDFAs are an exception.

5.2.2 Equivalence of Transitions

A failure transition in one FDFA is said to have a one-to-one mapping onto a failure transition in another FDFA if the two transitions have the same source state and same destination states in the two FDFAs. Similarly, a symbol transition in one FDFA (or DFA) is said to have a one-to-one mapping with a symbol transition in another FDFA (or DFA) if the two transitions have the same source state and same destination states in the respective automata. Such failure or symbol transitions may be regarded as equivalent.

This subsection compares one-to-one transition mappings of the various DHA FDFA and D^2FA FDFAs against the AC-fail FDFAs. This exercise is conducted for general interest and to enhance the understanding of the character of the FDFAs. Both symbol transitions and failure transitions are measured for the degree of equivalence with AC-fail FDFAs.

Symbol Transition Inequivalence

It was seen above that in many cases the percentage transition reduction over AC-opt transitions by the various FDFA variants closely corresponds to

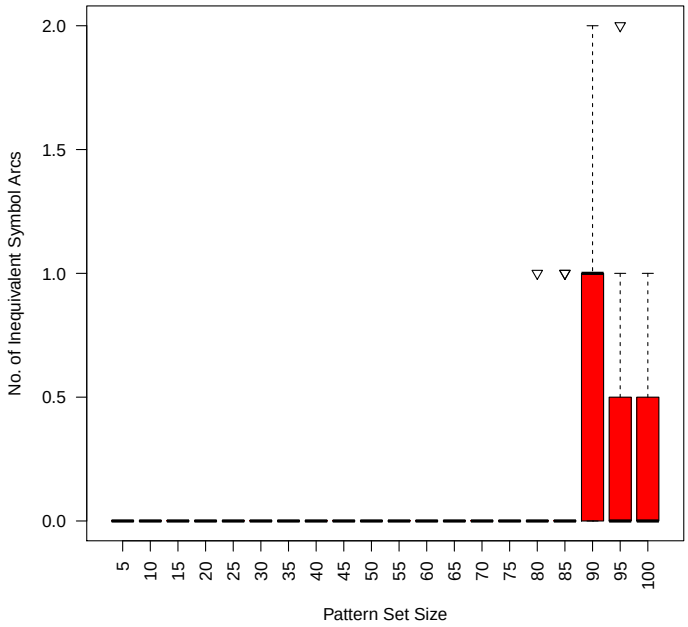

Figure 5.7: The *number of* non-equivalent *symbol* transitions between AC-fail FDFAs and three DHA FDFAs types (MaxIntent, MaxInt-MaxExt or MinExtent), $|\Sigma| = 10$.

that of AC-fail FDFAs. However, this does not necessarily mean that the *positioning* of failure and symbol transitions should show a one-to-one equivalence. Two symbol transitions δ and δ' from two automata are equivalent if and only if $\delta(q, a) = \delta'(q, a)$.

The extent to which the symbol transitions *do not* precisely match one another is shown in Figures 5.7 to 5.9 for $|\Sigma| = 10$ data sets and Figures 5.10 to 5.12 for $|\Sigma| = 4$ data sets. The figures provide box-whisker plots showing the number of non-equivalent symbol transitions between AC-fail FDFAs and other FDFAs. These box-whisker plots show explicitly the median, 25^{th} and 75^{th} percentiles as well as outliers of each of the 12 sample keyword sets of a given size.

As seen in Figure 5.7, the symbol transitions for the three DHA heuristics — MaxIntent, MinExtent and MaxInt-MaxInt — are practically identical to

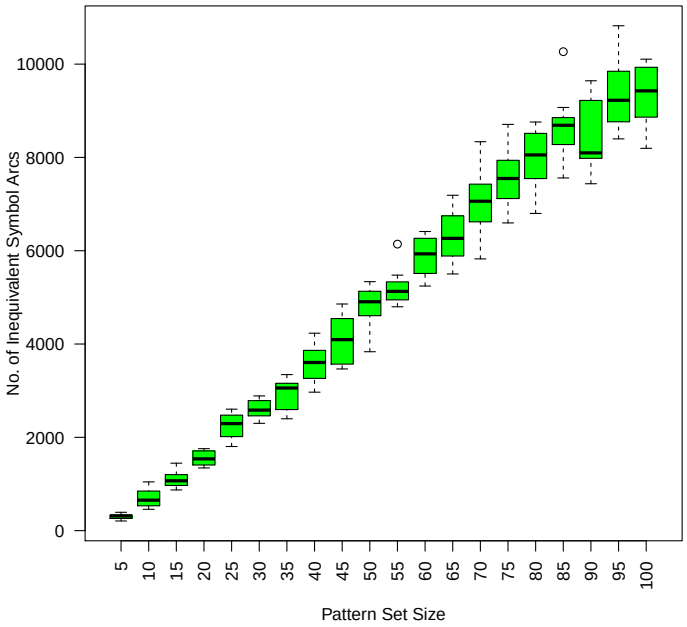

Figure 5.8: The number of non-equivalent *symbol* transitions between AC-fail FDFAs and MaxAR FDFA, $|\Sigma| = 10$.

those of AC-fail FDFA for $|\Sigma| = 10$ automata. Symbol transition matches are exact in these cases and occasional deviations from AC-fail FDFAs are only noted for pattern set sizes greater than 75. In the outlier cases, symbol transition disparity is at most two symbol transitions. This demonstrates that the aforementioned classes of DHA FDFAs are, in general, structurally nearly equivalent to AC-fail FDFAs' symbol transitions.

In these outlier cases, the data showed that the symbol transition size of AC-fail FDFAs is less than that of the DHA-FDFAs. AC-fail FDFA is the minimal FDFA[2] that can be obtained from AC-opt DFA.

Figure 5.8 and Figure 5.9 give box-whisker plots, for the $|\Sigma| = 10$ data, showing the number symbol inequivalences in the cases of MaxAR FDFAs and D^2FA FDFAs, respectively. The differences are approximately linearly

[2]By a minimal FDFA, we mean an FDFA that has removed the maximum possible number symbol transitions from the DFA.

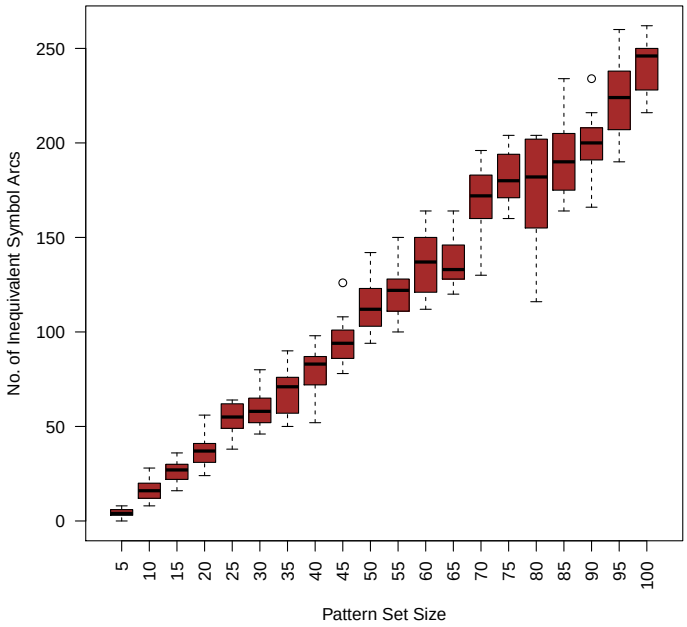

Figure 5.9: The *number of* non-equivalent *symbol* transitions between AC-fail FDFA and D^2FA, $|\Sigma| = 10$.

dependent on the size of the keyword set, reaching over 9000 for MaxAR FDFAs and approximately 250 for D^2FA FDFAs when the keyword sets of size is 100.

Figures 5.10 to 5.12 give the same box whisker plots for $|\Sigma| = 4$ automata. Here the number of non-equivalent symbol transitions of the three DHA FDFAs (which exclude MaxAR FDFA) is considerably larger than when $|\Sigma| = 10$, but nevertheless still quite modest. Ignoring the outlier cases, the maximum number of symbol transition inequivalences rises to about 20. In respect of the MaxAR FDFAs and D^2FA FDFAs, Figures 5.11 to 5.12 show that the number of inequivalent transitions remains more or less linear with the pattern set size, rising to 2000 for MaxAR FDFAs and reaching over 400 for D^2FA FDFAs.

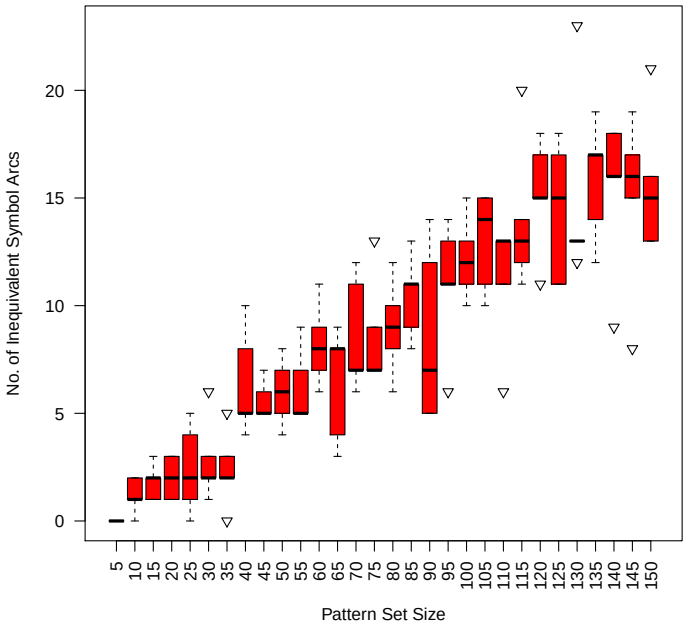

Figure 5.10: The *number of* non-equivalent *symbol* transitions between AC-fail FDFAs and three DHA FDFAs types (MaxIntent, MaxInt-MaxExt or MinExtent), $|\Sigma| = 4$.

Failure Transition Equivalences

This investigation explores the equivalences of the FDFA failure transitions, using the AC-fail FDFA failure transitions as the reference. Correspondence of both the source state and target states of failure transitions are checked between AC-fail FDFAs and other FDFA types. Figures 5.13 to 5.15 portray the percentages of equivalent failure transitions between the AC-fail FDFAs and the other FDFA types for $|\Sigma| = 10$ automata. Figures 5.16 to 5.18 show similar results for $|\Sigma| = 4$ data sets.

For the $|\Sigma| = 10$ data sets, the diagrams for failure transition differences in regard to MaxIntent FDFA, MinIntent-MaxExtent FDFA and MinExtent FDFA show directly resembling images and thus the FDFAs are fully equivalent. As seen in Figure 5.16, only in isolated instances they are 100%

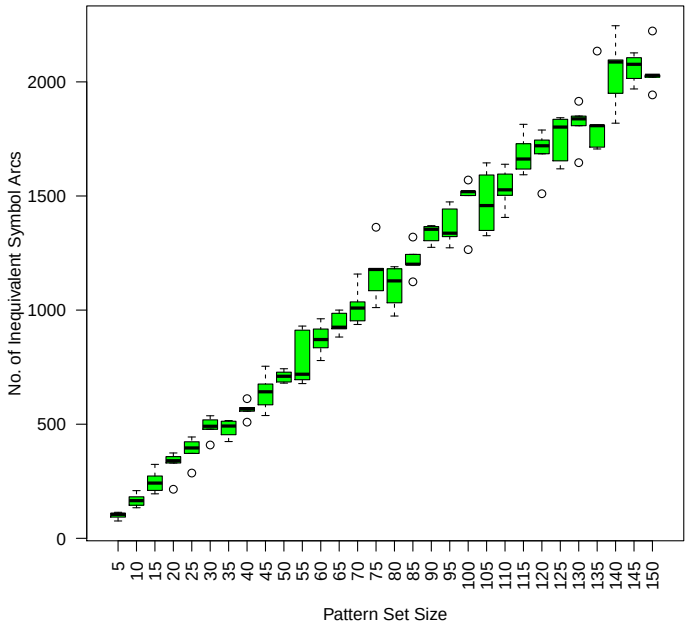

Figure 5.11: The *number of* non-equivalent *symbol* transitions between AC-fail FDFAs and MaxAR FDFA, $|\Sigma| = 4$.

equivalent to those of AC-fail FDFAs. The equivalence levels range from a median value of slightly more than 95% in the case of the smallest keywords sets to a median of about 50% for the largest keyword sets. For FDFAs with pattern set sizes up to 65, the median equivalence levels are 75% and above. At pattern set sizes greater than 65, the results decline gradually, in some cases falling below 40%.

Figure 5.14 shows that, in the case of MaxAR FDFA using the $|\Sigma| = 10$ data sets, the extent of equivalence is much lower. The median value percentage equivalence is, at best, just over 60% for small keyword sets, dropping close to zero for medium range keyword set sizes, and then increasing slightly to about 10% for the largest sized keyword sets.

Figure 5.15 gives, for the $|\Sigma| = 10$ data sets, the equivalence levels of D^2FA FDFA failure transitions compared to AC-fail FDFAs. The figure

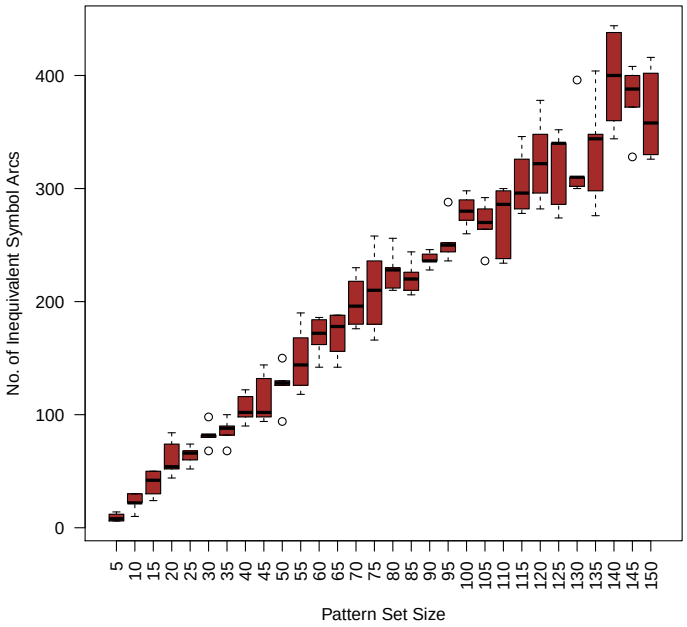

Figure 5.12: The *number of* non-equivalent *symbol* transitions between AC-fail FDFA and D^2FA, $|\Sigma| = 4$.

shows that the median percentage equivalence is in the range of about 15 - 25% for pattern set sizes up to about 80, and tends to drop to between 10 - 15% as the pattern set size becomes larger.

Now we switch to observations with respect to automata built from an alphabet size $|\Sigma| = 4$. (See Figures 5.16 to 5.18.) Figure 5.16 shows the results with respect to the DHA FDFAs (excluding MaxAR). The results are seen to be similar to $|\Sigma| = 10$ case. The median values are between about 75% - 90% for the data sets whose pattern set sizes are between 5 and 65. For larger pattern set sizes, the median results drop, eventually declining to below 40% for pattern set sizes of about 130 or larger. For the MaxAR heuristic data sets, the median correspondence is between 11% and 40% (see Figure 5.17). Meanwhile in Figure 5.18, for the D^2FA FDFAs the highest AC-fail FDFA failure transition equivalence median is just over 20% (for data based on pattern set sizes of 35).

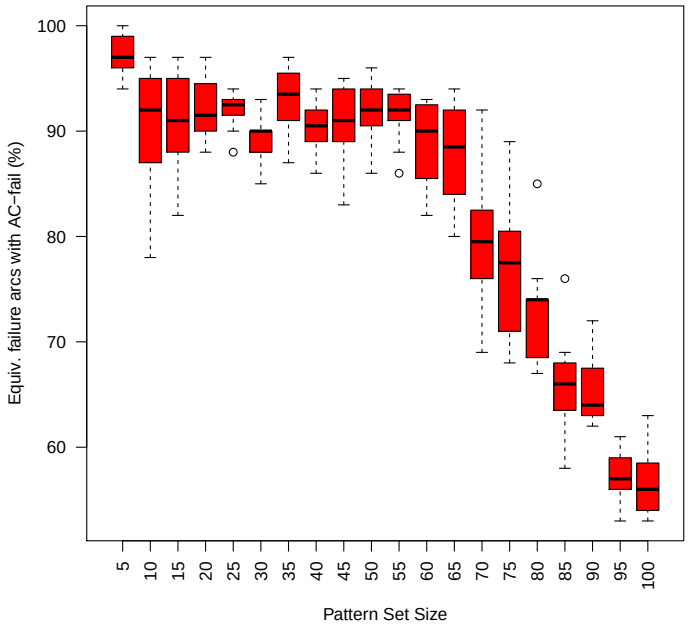

Figure 5.13: Equivalent *failure* transitions between AC-fail FDFAs and three DHA FDFAs (MaxIntent, MinExtent or MaxInt-MaxExt) in percentages of $|\mathfrak{f}|$ per pattern set, $|\Sigma| = 10$.

5.3 An Analysis of the Results

Overall, Section 5.2 reveals that there is a variety of ways in which failure transitions may be positioned in an FDFA that lead to very good or in many cases even optimal transition reductions. This is reflected in the data for all the FDFA types other than those FDFAs based on the MaxAR heuristic. It is interesting to note that even for the D^2FA FDFAs, the total number of transition reductions is very close to optimal, despite relatively large differences in the positioning of the transitions. However, the results also show that this flexibility in positioning failure transitions to achieve a good reduction in the number of transitions eventually breaks down, as in the case of the MaxAR FDFAs.

The heuristics used to generate FDFAs could be ranked in descending or-

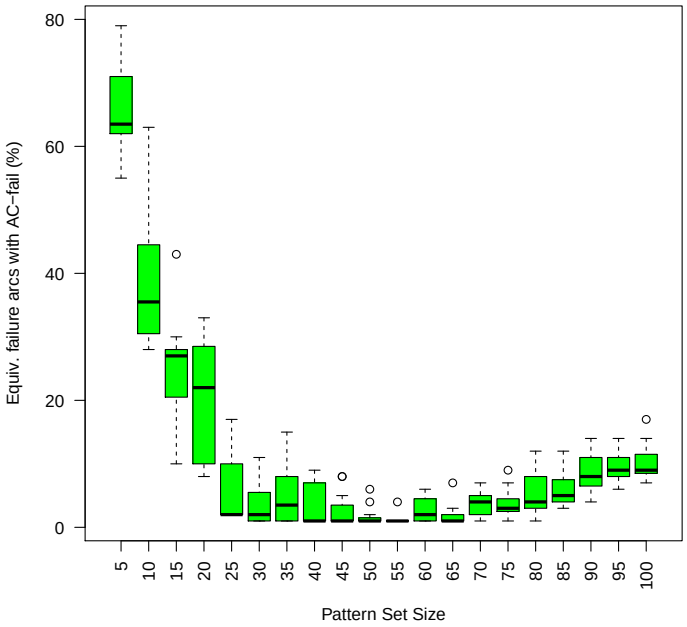

Figure 5.14: Equivalent *failure* transitions between AC-fail FDFA and DHA MaxAR FDFA in percentages of AC-fail $|\mathfrak{f}|$ per pattern set, $|\Sigma| = 10$.

der as follows to reflect the extent to which their resulting FDFAs correspond to their AC-fail FDFA counterparts:

$$\begin{aligned} MaxIntent\_FDFA &= MaxIntentMaxExtent\_FDFA \\ &= MinExtent\_FDFA \\ &> D^2FA\_FDFA \\ &> MaxAR\_FDFA \end{aligned}$$

The two MaxIntent based FDFAs and MinExtent FDFA most closely matched the AC-fail FDFA failure transitions. MaxAR FDFAs did not provide desirable results. Since the original focus of this study was to explore heuristics for the DHA, further comments about the D^2FA algorithm are reserved for the general conclusions in Chapter 7.

It was noted that the average percentage of symbol transitions removed

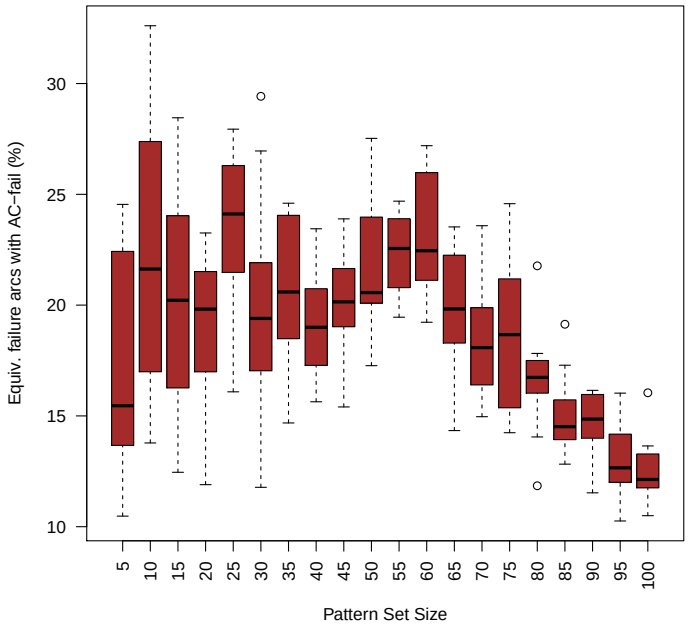

Figure 5.15: Equivalent *failure* transitions between AC-fail FDFAs and D^2FA in percentages of AC-fail $|\mathfrak{f}|$ per pattern set, $|\Sigma| = 10$.

for $|\Sigma| = 4$ data was about of 75%, whereas for $\Sigma = 10$ data it was about 90%. It is no surprise that the transition reduction is larger as the size of the alphabet increases. It is in the nature of the problem domain that as the alphabet set increases for a given number of patterns, the opportunity increases to remove symbol transitions and replace them by a single failure transition. Theoretically, the observed results conform to Proposition 4.1.1 - Equation 4.1, which estimates the symbol transitions size of AC-fail FDFAs to be $|\delta| \approx |\Sigma| + |Q|$. Assuming $|\Sigma| = n$ where $n \in \mathbb{N}$, then up to n transitions can be removed from each state, irrespective of the number of states $|Q|$. Hence, with larger alphabet sets, for example $|\Sigma| = 128$, the transitions removed from a DFA's transitions may approach 100% (as shown by Proposition 4.1.1). This explains why Kumar et al. [12] obtained up to 95% transition reduction with an ASCII symbol set using D^2FA algorithm to generate failure automata.

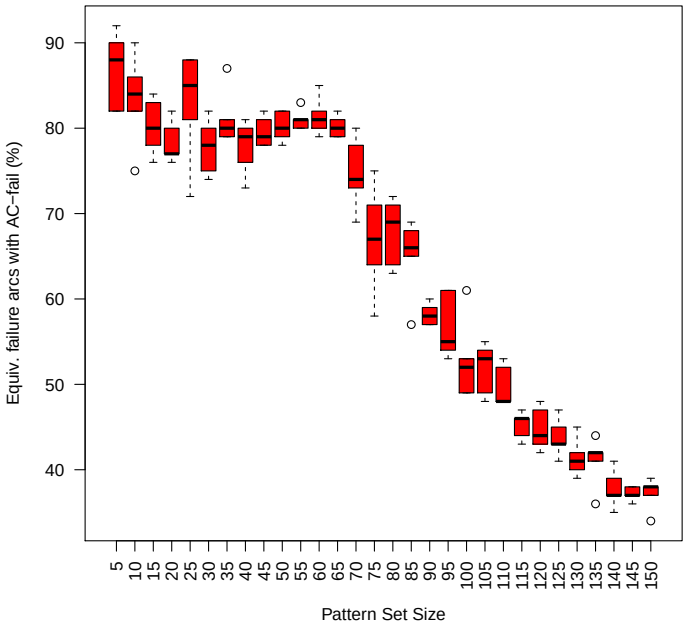

Figure 5.16: Equivalent *failure* transitions between AC-fail FDFAs and three DHA FDFAs (MaxIntent, MinExtent or MaxInt-MaxExt) in percentages of AC-fail $|\mathfrak{f}|$ per pattern set, $|\Sigma| = 4$.

All the heuristics used in the DHA approach are greedy: the heuristic is used to make a selection that appears best immediately, without considering how this "greedy" choice might affect a subsequent selection. Such greedy strategies are not guaranteed to produce optimal results. Nevertheless, it was clearly seen that MaxIntent FDFAs, MinExtent and MaxIntent-MaxExtent FDFAs practically reproduced AC-fail FDFAs in respect of symbol transitions, despite their greedy nature.

On the other hand, the MaxAR heuristic failed in removing large numbers of symbol transitions, paying an apparent price for following this opportunistic selection strategy.

The rationale for the MaxAR heuristic is clear: it will cause the maximum reduction in transitions in a given iteration. It was in fact the initial criterion proposed by Kourie et al. [9]. It is therefore somewhat surprising that it did

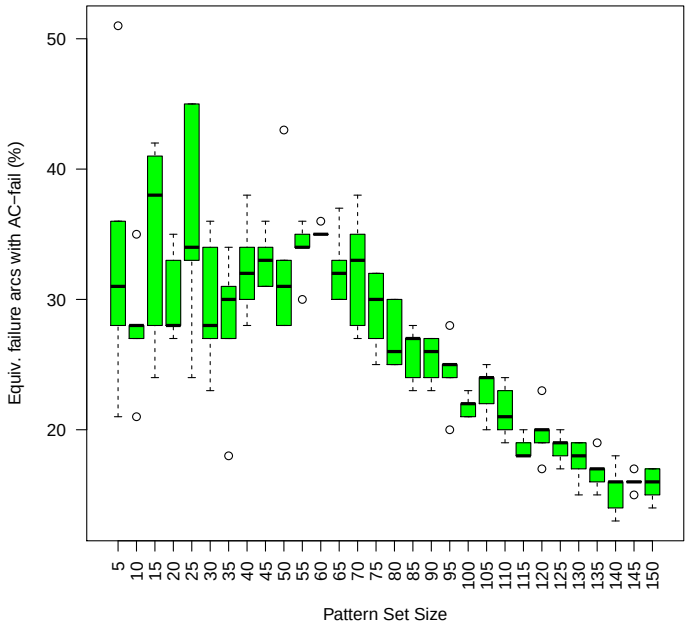

Figure 5.17: Equivalent *failure* transitions between AC-fail FDFA and DHA MaxAR FDFA in percentages of AC-fail $|\mathfrak{f}|$ per pattern set, $|\Sigma| = 4$.

not perform very well in comparison to other heuristics. It would seem that, in the present context, it is too greedy — i.e. by selecting a concept whose extent contains the set of states that can effect maximal reduction that in one iteration, it unfavourably eliminates from consideration concepts whose extent contain some of those states in subsequent iterations. Note that, being based on the maximum of the product of extent and intent sizes, it will tend to select concepts in the middle of the concept lattice diagram.

It was only when early trials in our data showed up the MaxAR heuristic's relatively poor performance, that the MaxIntent, MaxIntent-MaxExtent and MinExtent heuristics were introduced. These heuristics prioritise concepts in the top or bottom regions, respectively, of the line diagram of the concept lattice. The MaxIntent heuristics maximise the number of symbol transitions to be removed *per state* when replacing them with failure transitions (as opposed to the MaxAR heuristic which accounts for removal of symbol

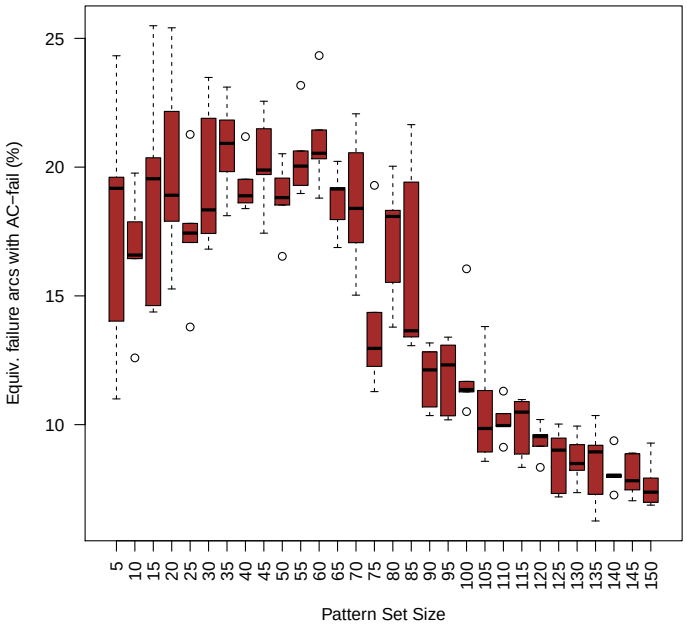

Figure 5.18: Equivalent *failure* transitions between AC-fail FDFAs and D2FA in percentages of AC-fail $|\mathfrak{f}|$ per pattern set, $|\Sigma| = 4$.

transitions across a *set of states*). By the partial order characteristic of concepts, concepts with large intents tend to have small extents and *vice-versa*. For this reason, the MinExt based heuristics are likely to select the same concepts as the MaxInt heuristic when executing the DHA. This explains why the results are similar, though not identical.

The difference between the MinExtent and MaxIntent heuristics lies in the selection order of concepts during the iterations of DHA. For example, a first chosen concept using the MaxIntent heuristic may be the n^{th} chosen concept using the MinExtent heuristic.

The MaxIntent-MaxExtent heuristic is a specialization of the MaxIntent heuristic, with MaxInt-MaxExt heuristic implementing a secondary priority criterion on the maximum extent. This explains the close correspondence between their resulting FDFAs. Because of the close correspondence in the

FDFAs they produce, we conjecture for the most part, the same concepts are selected by these two heuristics to form their respective FDFAs. This conjecture requires further study. Although such a relationship is, of course, data-dependent, random data tends in that direction, as was confirmed by inspection of our data — resulting in similar graphs for the latter three heuristics.

The occurrence of "useless" failure transitions is one of the reasons for differences between AC-fail FDFAs and the DHA FDFAs. Such failure transitions are, of course, redundant. The graphs in Figures 5.3 and 5.4 showed that some of the randomly generated keyword sets lead to "useless" failure transitions, but they are so rare that they do not materially affect the overall observations.

Aside from MaxAR, then, these various heuristics appear to be rather successful at attaining AC-fail-like FDFAs. However, the *ClosestToRoot* heuristic (implemented as *StateWithLeastDepth* variant for input AC-opt DFA) has also played a part in this success.

It is worth reflecting briefly on the way in which the original AC-fail algorithm (as implemented by SPARE Parts) works. In that algorithm, the AC-fail FDFA failure transitions are designed to record that a suffix of a state's string is also a prefix of some other state's string. Thus, $\mathfrak{f}(q) = p$ means that a suffix of state q's string is also a prefix of state p's string. However, since there may be several suffixes of q's string and several states whose prefixes meet this criterion, the definition of f requires that the *longest possible* suffix of q's string should be used. This ensures that there is only one possible state, p, in the trie whose prefix corresponds to that suffix. Thus, on the one hand, AC-fail algorithm directs a failure transition "backwards" towards a state whose depth is less than that of the current state. Put differently, the AC-fail algorithm selects a failure transition's target state to be as *far* as possible from the start state, because the suffix (and therefore also the prefix) used must be maximal in length.

The *ClosestToRoot* heuristic approximates the AC-fail FDFA action in

that it also directs failure transitions backwards towards the start state. However, by selecting a failure transition's target state to be as *close* as possible to the start state, it seems to contradict AC-fail algorithm actions. It is interesting to note in Figures 5.13 and 5.16 that MinExtent FDFA, MaxIntent FDFA and MaxInt-MaxExt FDFA show a rapid and more or less linear decline in failure transition equivalence with respect to AC-fail FDFAs when pattern set size reaches about 65. We conjecture that for smaller keyword sizes, the *ClosestToRoot* heuristic does not conflict significantly with AC-fail algorithm's actions because there is little to choose in the backward direction; and that when keyword set sizes become greater, there is more choice, and consequently less correspondence between the failure transitions. This is but one of several matters that has been left for further study.

5.4 Conclusion

Empirical results of the transition comparison of the various failure automata against Aho-Corasick automata were presented and analysed. Generally, the failure automata were impressive with their transition reduction. An exception was MaxAR FDFA.

Inductively, based on the outcomes of this chapter's Aho-Corasick automata specific experimental findings, in the next chapter we apply the same modifications to the DFA-homomorphic algorithm to produce FDFAs in the general case out of 'random' DFAs. Similar to the current chapter, transition saving measurements are recorded and analysed.

Chapter 6

Transition Reductions in the General Case

6.1 Introduction

In Chapter 5 we found that some variants of DHA optimally removed symbol transitions of AC-opt DFAs when generating FDFAs — i.e. the FDFAs produced by these variants resembled their AC-fail FDFA counterparts. The variants that performed the best were based on the heuristics we called MaxIntent, MaxInt-MaxExt and MinExtent. The *ClosestToRoot* heuristic that was used for selecting the target state of a failure transition was referred to as the *StateWithLeastDepth* implementation. We found that both the symbol transitions and the failure transitions of the FDFAs resulting from aforementioned DHA variants that relied on these heuristics corresponded very closely to those produced by the AC-fail FDFA. This comparison of DHA FDFAs against AC-fail FDFAs outlined in the previous chapter is a concrete but only a limited starting point for assessing the performance of DHA.

The goal of this chapter is to investigate the effect on transition reductions by the various DHA-related heuristics previously discussed in Chapter 3 for building language-equivalent FDFAs from *general* complete DFAs. We con-

tinue to consider the MaxIntent, MaxInt-MaxExt, MinExtent and MaxAR heuristics. The D^2FA algorithm also formed part of this investigation.

Instead of using the trie specific *StateWithLeastDepth* criterion for selecting the target state of failure transitions in DHA (which was used in the previous chapter), the *ClosestToRoot* heuristic will be generalised by using the *StateWithLeastDistanceFromStartState* criterion. This criterion relies on Dijkstra's algorithm for calculating the shortest distance from the start state to each DFA state. These heuristics were described in Chapter 3.

This study is confronted by the following dilemma. In this domain it is known *a priori* that it is NP-complete to determine whether the a given FDFA is optimal in the sense of having the fewest possible transitions while remaining language-equivalent to the original DFA. This was proved by Björklund et al. [24]. This means that, starting with some randomly generated complete DFA, there is no evident way of deciding how well or badly DHA has performed in generating a language-equivalent FDFA.

Moreover, generating DHA FDFAs from DFAs that have been randomly generated may not result in interesting transition reductions, simply because such randomly generated DFAs do not offer much scope for introducing failure transitions in the first place. Our early experiments along these lines showed that the percentage reduction in transitions of DHA FDFAs from random DFAs was generally less than 10%. An alternative way of generating "interesting" DFA test data was therefore sought.

Our approach begins with generating "random FDFAs"[1]. Using such an FDFA we then create a language equivalent DFA that, in turn, serves as an input for DHA. By this approach, we at least have a basis for comparing the performance of DHA, in that we can compare the number of transitions in the generated DHA FDFA against the original FDFA that was used to create the input DFA.

[1] The FDFAs are not entirely generated randomly, some FDFA properties are considered when inserting random transitions. The properties will be discussed in the following section.

6.2 Generating 'Random' FDFAs and DFAs

This section describes an algorithm that constructs random FDFAs and random DFAs. These are the principal data elements for the experiment. Firstly, a short literature survey about randomly generated automata is presented. Then the algorithm is laid out and discussed. Lastly, an example is presented that demonstrates how the algorithm constructs an FDFA and its DFA counterpart.

6.2.1 Random Automata Related Studies

There has been much work on generating random DFAs in the literature. Research on random DFAs is actively conducted in the study of the *computational learning theory*, specifically from the topic of *learning a DFA*. The learning DFA problem is generally known to be computationally hard [61, 62].

In learning DFAs the following topics are studied about random DFAs: the probability distribution on sample strings read by the random DFA [63], state complexity of a random DFA [64], random walks (the probability of transition path traversal) on random DFAs [62], algorithms for learning random DFAs [65] and their complexities, and other related topics. Nicaud [66] gives a survey of some of the recent studies in this regard. However, as pointed out above, the interest in this study is not so much in generating random DFAs as test data. This is because generating FDFAs from random DFAs results in very small transition reductions. For this reason, there will be no further consideration given here to generating random DFAs for test data.

A typical randomly generated DFA $\mathcal{D} = (Q, \Sigma, \delta', F, q_s)$ must have the following properties to allow uniformly distributed samples of the DFA structure [62, 64, 65]. A single state is chosen randomly over Q to be the initial state, q_s. Because of the symmetry property, Berend and Kontorovich [64]

suggested that the choice of the start state, q_s, can be simplified by setting q_s to be the first state from a list of states — e.g. set $q_s = q_0$, where $Q = \{q_0, q_1, \ldots, q_n\}$. In order to determine final state set F, we uniformly assign binary values (either 0 or 1) to each state of Q. For the transitions set δ made by the product $\Sigma \times Q$, all the entries are also randomly assigned to values in Q. All the above random choices are mutually independent.

In Björklund et al. [24, 10], the following observation is made about building an FDFA from a DFA.

Property 6.2.1. *Given an arbitrary FDFA, an equivalent DFA with the equal number of states can be built in polynomial time.*

The proof of this observation (also provided by [24, 10]), reproduced below, forms the base for us to develop an algorithm that generates a random FDFA whilst concurrently building a language equivalent DFA.

Proof. Given an FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, q_s)$, we construct an equivalent DFA $\mathcal{D} = (Q, \Sigma, \delta', F, q_s)$ as follows. Clearly, we see that every part of $\mathcal{D}$ is in $\mathcal{F}$ save for the differences in δ and δ'. Firstly, we assign $\delta' = \delta$. Afterwards, we process the states in Q, possibly by adding outgoing transitions. If $q_1 \in Q$ has no failure transitions in $\mathfrak{f}$, the out-transitions from q_1 remain unchanged. If q_1 has a failure transition, allow $q_1, q_2, \ldots q_k$ be the failure path from q_1 i.e. build the composite failure transition functions such that $q_2 = \mathfrak{f}(q_1), q_3 = \mathfrak{f}(\mathfrak{f}(q_1))$ and so on and so forth. Note that the failure path is unique since $\mathfrak{f}$ is a function. If the failure path forms a failure cycle, then let q_k be the last state before the cycle closes (to avoid a divergent failure cycle). We view the states on the path in some order, starting with state q_2. When some q_i (from the failure path) is reached then for each $a \in \Sigma$ such that the q_1 does not yet have an outgoing transition on a in δ', and such that there exists some $q_j \in Q$ with $\delta(q_i, a) = q_j$, we set $\delta'(q_i, a) = q_j$. □

We use some details from the above proof to generate a random FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, q_s)$ and alongside it, a language equivalent DFA $\mathcal{D} = (Q, \Sigma, \delta', F, q_s)$. In our case we build transitions of the two language equivalent automata in some random manner as it will be shown in the upcoming sections.

6.2.2 The "Random" (F)DFA Algorithm

This algorithm (see Algorithm 6.1) constructs both a "random" FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, q_s)$ and its language equivalent DFA $\mathcal{D} = (Q, \Sigma, \delta', F, q_s)$. This algorithm is designed as part of this dissertation specifically for use in Section 6.3. The two automata have the same sets of states, Q; a corresponding start state q_s; and equivalent sets of final states, F. These attributes for the pair of automata are assumed to have been defined before the algorithm is invoked. Only the transitions: δ, δ' and $\mathfrak{f}$ are not yet fully defined when the algorithm starts executing.

Variables for Transition Functions

The transition functions (δ' for the DFA, and δ and $\mathfrak{f}$ for the FDFA) are represented by global variables. We assume that the assignment operation establishes the function value for the given parameters. For example, $\delta(q_i, a) := q_j$ means that the value q_j is assigned to the function δ for the parameter values (q_i, a). The δ and δ' functions can be implemented as 2D tables. The foregoing assignment fills the value q_j in the cell whose row represents state q_i and whose column represents symbol a. Similarly, $\mathfrak{f}$ may be represented by a 1D array. For example, for a failure transition set by the assignment $\mathfrak{f}(q_i) := q_j$, we store q_j into the array entry representing q_i and is indexed by i.

A desirable postcondition for the algorithm is that it should deliver automata whose states are all connected to the start state. The algorithm is designed so that if it is invoked with values already assigned to δ' (and the

Algorithm 6.1 The Random (F)DFA Algorithm

```
{ pre k ∈ [0, |Q|) }
proc (k)
    for each (q_i ∈ Q) →
        for each (a ∈ Σ) →
            if (δ'(q_i, a) ≠ ⊥) → skip
            [] (δ'(q_i, a) = ⊥) →
               h := q_i;
               T, l := {h}, random([0, k]);
               { Create a failure path starting from q_i of length ≤ l. }
               { h is head of this failure path to date }
               { Path states are stored in T }
               do ((|T| < l + 1) ∧ (δ'(h, a) = ⊥) ∧ f(h) ∉ T) →
                    if (f(h) = ⊥) → f(h) := random(Q \ T);
                    [] (f(h) ≠ ⊥) → skip
                    fi;
                    h := f(h);
                    T := T ∪ {h};
               od;
               { (|T| ≥ l + 1 ∨ δ'(h, a) ≠ ⊥ ∨ f(h) ∈ T) }
               { If δ'(h, a) is undefined, then find random value }
               if (δ'(h, a) = ⊥) → δ'(h, a) := random(Q);
               [] (δ'(h, a) ≠ ⊥) → skip
               fi;
               δ(h, a) := δ'(h, a);
               for (q_k ∈ T) →
                    δ'(q_k, a) := δ(h, a);
               rof
            fi
        rof
    rof
corp
{ post Complete D ∧ L(D) = L(F) }
```

same values to δ) that ensures this connectivity, then the connectivity will be retained when the algorithm terminates. The algorithm terminates when *all* states of the resulting DFA have a symbol transition to some other state (i.e. when δ' is such that the DFA is "complete"). It ensures that upon termination δ and $\mathfrak{f}$ for the FDFA are such that the FDFA defines the same language as the DFA.

The connection of transitions can be implemented as follows. For each $i < (|Q| - 1)$, a randomly selected $a \in \Sigma$ is drawn and we ensure that $\delta(q_i, a) = \delta'(q_i, a) = q_{i+1}$. That is, the transition $\langle\langle q_i, a\rangle, q_{i+1}\rangle$ is added to both sets δ and δ'. (Assume that q_0 is taken as the start state, q_s.)

The algorithm starts with some (or all) values of δ', δ and $\mathfrak{f}$ undefined. It terminates with all values of δ' defined and some (or all) values of both δ and $\mathfrak{f}$ defined. The symbol $\perp$ is used to denote a so-called *invalid or undefined* transition — e.g. $\delta(q_i, a) = \perp$ means that there is no symbol transition on a in state q_i. Thus, if such a transition is not encountered at state q_i, then a failure transition should be taken. It is also possible for an FDFA to have undefined failure transitions at one or more states. If $\mathfrak{f}(q_i) = \perp$, this means that there is no failure transition exiting state q_i.

Other Variables

To influence the number of failure transitions in the final FDFA, the algorithm is invoked with an input parameter $k \in \mathbb{N} \wedge k < |Q|$, whose value is to be pre-selected. The integer k serves an upper bound on the length of any failure path that may appear in the FDFA. The role of k is thus to constrain the number of failure transitions in the FDFA that need to be traversed before a symbol transition on any given symbol is encountered. Hence, when $k = 0$ the result will be a degenerate FDFA (i.e. $\mathcal{F} = \mathcal{D}$). Note that, because of various random features built into the algorithm, the extent to which the value of k influences the number of failure transitions is not known or predictable *a priori*. All that can be said is that the larger the value of k, the fewer symbol transitions and the more failure transitions are likely to be

present in the final FDFA.

While k applies globally, whenever a given state, q_i and a symbol a, are to be considered, two local parameters are used: the variable l and the set variable T.

We first indicate how l is used. This parameter specifies the maximum distance along a failure path to travel from state q_i until a state that provides a symbol transition on a is encountered. The variable $l \in \mathbb{N}$ must be defined in the range $[0, \ldots, k]$. Similarly to k, the variable l controls the potential maximum failure transition path to traverse in order to consume the symbol a when processing strings.

The role of l is explained as follows. When $l = 0$ and the symbol transition is not yet defined, there is no failure transition to insert from the current state. In the case when $l = 1$, a single failure transition may be taken from the current state to some other state, provided that the current state has an invalid symbol transition on the associated symbol. If $l = 2$ there must be at most two failure transitions taken to consume a, and if $l = 3$ then a maximum of three failure transitions to may be taken to consume the letter, and so on.

The local variable T of the algorithm is of "set" type and recomputed for each combination of state q_i and symbol a that is considered. The set T is used to accumulate states of the failure path to follow, starting from state q_i, and at whose head state the will be a symbol transition on a. The set T serves two purposes. Firstly, by keeping track of all states of the failure path one can avoid constructing divergent failure cycles. In addition T is also used to determine how to direct symbol transitions on a in the various DFA states.

Algorithm 6.1 gives the pseudocode. It builds the DFA $\mathcal{D} = (Q, \Sigma, \delta', F, s)$ and language-equivalent FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ by assigning values to δ', δ and $\mathfrak{f}$ without changing those values already assigned before invoking the algorithm. In order to insert transitions, a function named *random* is assumed that randomly selects an element from the set that is given as its

parameter. For each state, the algorithm iterates over all possible alphabet symbols. The algorithm is described in further detail below.

Inserting Transitions

The total number of the symbol transition table entries to be considered for the FDFA's symbol transition function δ is $(|Q| \times |\Sigma|)$, some of which will be valid and others, invalid. The table for symbol transition function, δ', of the complete DFA is the same size, but because the DFA is complete all its entries have to be valid. To determine the symbol transition table entries for each function, the algorithm iterates over each state (i.e. row of the transition table) and over all possible alphabet symbols (i.e. column of the transition table).

For a given $a \in \Sigma$ and $q_i \in Q$, the following *four* steps are executed by the algorithm:

1. Firstly, the algorithm decides whether it is necessary to insert an entry at the current transition $\delta'(q_i, a)$ or not. If there is already a valid symbol transition at $\delta'(q_i, a)$ then there is no need to insert a symbol transition and the algorithm moves on to execute the next transition; otherwise steps 2, 3, and 4 are undertaken.

2. Secondly, the local variables h, l and T are initialized. The variable h is used to represent the state at the head of a failure path that is under construction in the algorithm. At this stage, therefore, it is set to the current state, q_i. The integer l is initialized by assigning a value from the range $[0, k]$. The set T is also initialized by inserting state h as its initial single element.

3. Thirdly, a loop is executed that builds a failure path of length l or less in the FDFA.

4. Finally, symbol transitions into the FDFA and DFA are provided.

Step 3 is further elaborated in the next paragraph and then step 4 is described

in more detail.

The loop's condition consists of three conjunct predicates to guide the path of failure transitions starting from q_i. The three conjuncts described below are:

- $\delta'(h, a) = \perp$: In the DFA under construction, the symbol transition $\delta'(h, a)$ must not yet have been assigned if we are to use the state h as the current head of an FDFA failure path.
- $|T| < (l + 1)$: Note that in general there are $x + 1$ states in a failure path of length x. The number of states in the failure path, $|T|$, must therefore be strictly less than the integer $l + 1$ for the loop to execute and add another state into the failure path. If there are already $l + 1$ states in T, then we have reached the size limit of the failure path and the loop should not be entered.
- $\mathfrak{f}(h) \notin T$: This conjunct ensures that the loop terminates if a failure path has been built that is in fact a failure cycle.

If all three conjuncts are true then a failure transition from h to a random target is created provided such a transition does not already exist. The algorithm randomly selects a state from a specific set of states described below. The failure transition target state is then inserted into the set T and the head of the failure path, h, is updated to this target state.

The algorithm prevents the new head of the failure path from being a state that is already on the existing failure path. This is so because the parameter of the function *random* specifally excludes states in T. Note, however, that it is possible that the newly selected head of the failure path h is the source of a failure transition whose target is a state in T. In such a case, the states collected in T constitute a failure cycle. We will argue below that any so-formed failure cycle is nevertheless not a divergent failure cycle.

Once the above loop terminates, the algorithm checks if the DFA's $\delta'(h, a)$ currently has a valid value. If not, then it inserts a randomly generated symbol transition destination for $\delta'(h, a)$. The transition value $\delta'(h, a)$ is

copied to its FDFA counterpart, $\delta(h, a)$. Additionally, all states in the set T have their symbol transition destinations on the symbol a updated to the same value as $\delta(h, a)$. Then the algorithm executes an iteration based on the next state / symbol pair to be considered.

In summary, the algorithm first selects a random number, $l \in [0, k]$. Then, the entries in the failure function (starting from the start state) are filled up to generate a failure path consisting of randomly selected states. The failure path length is maximally l and the failure path may possibly form a failure cycle. The above steps are repeated, ensuring at each iteration step that a state has maximally one exiting failure transition. When it is no longer possible to generate any more failure paths of length l or less, then fill in the rest of the transition table in some random fashion. At this stage, a "random" FDFA is generated. And finally, the relevant FDFA transitions are used to update the DFA transitions in an orderly manner.

Note that even though the finally obtained FDFA may contain failure cycles, no *divergent* failure cycles can be generated. To see this, recall that a failure cycle is not divergent if and only if for every $a \in \Sigma$, there is at least one state in the cycle, say q_k, such that $\delta(q_k, a) \neq \perp$. To prove the non-divergent nature of any cycle that may be formed by the algorithm, it is sufficient to consider the outcome at the end of each iteration of the inner for-loop (over alphabet symbols). It is clear that for the state $q_i \in Q$ and $a \in \Sigma$ under consideration at that point, a failure path would have been generated of maximum length l. The head of that failure path is denoted by state h in the algorithm. The algorithm ensures that $\delta(h, a)$ has a valid value when the end of the loop in question is reached. Since this holds for a given $q_i \in Q$ for every iteration of the for-each loop over $a \in \Sigma$, it is guaranteed that on a (possibly degenerate) failure path leading from q_i there will be some state, h, such that $\delta(h, a)$ has a valid value. And since this claim is also true for every $q_i \in Q$, it must be the case that all failure cycles are non-divergent.

Table 6.1: An example: creating a 'random' FDFA and a 'random' DFA.

(a) Initialized DFA

Q/Σ	a	b	c
0	1	⊥	⊥
1	⊥	2	⊥
2	⊥	⊥	3
3	⊥	⊥	⊥

(b) Initialized FDFA

Q/Σ	a	b	c	ƒ
0	1	⊥	⊥	⊥
1	⊥	2	⊥	⊥
2	⊥	⊥	3	⊥
3	⊥	⊥	⊥	⊥

(c) DFA: After $q_i = 0$ entries

Q/Σ	a	b	c
0	1	1	3
1	⊥	2	⊥
2	⊥	1	3
3	⊥	1	3

(d) FDFA After $q_i = 0$ entries

Q/Σ	a	b	c	ƒ
0	1	⊥	⊥	3
1	⊥	2	⊥	⊥
2	⊥	1	3	⊥
3	⊥	⊥	⊥	2

(e) DFA: After $q_i = 1$ entries

Q/Σ	a	b	c
0	1	1	3
1	1	2	3
2	⊥	1	3
3	⊥	1	3

(f) FDFA: After $q_i = 1$ entries

Q/Σ	a	b	c	ƒ
0	1	⊥	3	3
1	⊥	2	⊥	0
2	⊥	1	3	⊥
3	⊥	⊥	⊥	2

(g) DFA: After $q_i = 2$ entries

Q/Σ	a	b	c
0	1	1	3
1	1	2	3
2	2	1	3
3	⊥	1	3

(h) FDFA: After $q_i = 2$ entries

Q/Σ	a	b	c	ƒ
0	1	⊥	3	3
1	⊥	2	⊥	0
2	2	1	3	⊥
3	⊥	⊥	⊥	2

(i) Finally; DFA, $q_i = 3$

Q/Σ	a	b	c
0	1	1	3
1	1	2	3
2	2	1	3
3	2	1	3

(j) Finally; FDFA, $q_i = 3$

Q/Σ	a	b	c	ƒ
0	1	⊥	3	3
1	⊥	2	⊥	0
2	2	1	3	⊥
3	⊥	⊥	⊥	2

6.2.3 An Example

Tables 6.1 (a)-(j) show how a random FDFA and a random DFA are constructed using the above stated algorithm. This example is for inserting transitions for two language equivalent automata defined by $Q = \{0, 1, 2, 3\}, \Sigma = \{a, b, c\}, q_s = 0, F = Q$. The assumed value for k is 3. Some transition entries for the FDFA's δ and $\mathfrak{f}$ as well as the DFA's δ' are to be added by the algorithm.

Note that in this example, a state is not represented by the letter q subscripted by an integer, say i, but by the integer i itself. The start state is represented by the integer 0. Furthermore, as is often the case in this domain, the transition functions δ and δ' are represented as transition tables whose cell entries represent destination states.

In Tables 6.1, the transition tables of δ' and δ are presented as the algorithm reaches different stages of execution. The pair of automata are presented side by side with the DFA on the left hand side and the FDFA on the right hand side. They both contain three columns for the symbols of the alphabet and four rows for the states. The FDFA table has an additional column labelled $\mathfrak{f}$ for failure transitions.

1. Firstly, to connect all the states, for each $i \in \{0 \ldots 2\}$ the DFA and FDFA are provided with symbol transitions such that $\delta(i, g) = \delta'(i, g) = i+1$. Note that g is randomly chosen from $\{a, b, c\}$. (See *Tables 6.1 (a) and (b).*) The remaining transition cells remain invalid. These transition values are recorded in the tables for δ and δ' before the algorithm is invoked. The remaining transition are inserted by the algorithm as described in the following paragraphs.

   The algorithm now iterates over all states (i.e. rows), in each iteration considering all alphabet symbols (columns).

2. Tables 6.1 (c)-(d) depicts the transition tables of the two automata after the algorithm has iterated over the initial state 0.

Starting with the column for symbol a, it is seen that the transitions: $\delta(0, a)$ and $\delta'(0, a)$ already have values (namely 1), so there is nothing more to do.

Thus, the next symbol, b, is considered. Since $\delta'(0, b)$ is not yet determined, a new value for l is determined randomly. Suppose this value is $l = 2$. This means that the algorithm needs to generate a path of failure transitions of maximum length 2. The algorithm adds the current state 0 to the set T of states on this failure path and it assigns the value h to be 0. State 0 is thus the origin of a failure transition that terminates at a randomly selected state, say 3 in this case. Thus 3 is entered into the $\mathfrak{f}$ column. Additionally, state 3 is added to set T and is set as the head h. To complete the creation of a failure path of length 2, the algorithm now moves to the row for state 3 and generates a new random destination for this failure transition. Suppose it is 2. It indicates this in column $\mathfrak{f}$ of row 3. It then adds 2 to the set T. The head h becomes 3. By now $|T| = (l + 1)$ so there are no additional transitions to be added. Finally, at the head of this failure path, in state 2, the algorithm creates a new random destination for a transition on symbol b. Suppose this is 1. It then sets both $\delta'(2, b)$ and $\delta(2, b)$ to 1. And then, based on $\delta(2, b) = 1$, it also sets $\delta'(j, b)$ to 1 for each state $j \in T$, namely for $\delta'(1, b)$ and $\delta'(3, b)$.

To complete the iteration for row 0, the algorithm now has to provide an entry for $\delta'(0, c)$. Again the algorithm randomly generates a value for l. Suppose it is $l = 2$ as before. T is reinitialised and set to store state 0. From state 0 a failure transition to the target state 3 has already been established. Since $l = 2$, we again have to follow a failure path of length 2 when encountering a symbol c in state 0. The algorithm thus moves one failure transition ahead to state 3. It also adds 3 to T and assigns h to be 3. A destination state for a failure transition starting state 3 had previously been generated, namely 2 and so state 2 is inserted into set T. Then 2 becomes h. The failure path of length 2 has now been reached in state 2 and an entry for $\delta(2, c)$ is required.

Again it randomly generates a destination state, say 3 and sets $\delta(2, c)$ and $\delta'(2, c)$ to 3. Following the same procedure as for symbol b, the algorithm sets $\delta'(2, c)$, $\delta'(0, c)$ and $\delta'(3, c)$ to 3.

3. Tables 6.1 (e)-(f) illustrate the two automata after the algorithm has inserted all transitions that arise when dealing with state 1.

   The algorithm is considering to insert a transition at $\delta'(1, a)$. Because this entry $\delta'(1, a)$ has an invalid value, a valid entry must be made. Suppose that in considering column a the random failure path length l turns out to be 1. As before, state 1 is stored in T and is assigned to the head h. Suppose a new failure transition destination 0 is generated randomly. The state 0 is added to T and the head h becomes 0. The failure path has reached the limit $l = 1$ and since a valid transition destination already exists for $\delta'(0, a)$, the addition of failure transitions is aborted. Since there is an established symbol transition $\delta'(0, a)$ namely 1 at the current control state h, the algorithm simply leaves it in place — i.e. it does not randomly generate a new FDFA symbol transition destination at 0 on symbol a. Then $\delta'(0, a)$ is copied into $\delta(0, a)$. Now for each state $j \in T$ the algorithm sets $\delta'(j, a)$ to 1. Thus, $\delta'(0, a)$ remains at 1 and $\delta'(1, a)$ is set to 1.

   The algorithm now considers transitions on symbol b in state 1. Since there is already a valid transition for $\delta'(1, b)$ (namely to 2), the algorithm simply continues to execute the next transition entry $\delta(1, c)$.

   Now column c for state 1 has to be considered, and this entry $\delta'(1, c)$ has an invalid transition value. Suppose that at this stage, the algorithm randomly generates $l = 1$. The algorithm adds state 1 to T and sets h to be 1. Recall that when dealing with column a for state 1, a failure transition to state 0 was created, so there is already a failure transition from state 1. The algorithm therefore adds state 0 to T and sets the head h to be state 0. Because $|T| = (l + 1)$, there are no further failure transition destinations to consider. The algorithm generates a random symbol transition at $\delta'(0, c)$. Suppose that the new transition is 3. The

value 3 is copied to $\delta(0, a)$. Thus as usual, all values of T are updated with 3 — i.e. $\delta'(0, c)$ and $\delta'(1, c)$ are all set to 3.

4. Tables 6.1 (g)-(h) show the status of the automata after all transitions from state 2 have been generated by the algorithm.

   We now look at the algorithm inserting a transition at $\delta'(2, a)$. This entry is invalid so transitions must be added to the pair of automata. Suppose a random value set for l is 0. The head h is initialized by the current state value 2 and the set T is also initialized by the same value 2. Since $l = 0$, there are no failure transitions to be inserted. A new symbol transition value is then randomly generated for $\delta(2, a)$. Let the generated value be 2. As usual the newly generated DFA transition is copied into the corresponding FDFA transition. Lastly, using the lone value in T, the δ' is updated such that is $\delta'(2, a)$ becomes 2.

   The algorithm then continues with symbol transition iterations at state 2, first to $\delta'(2, b)$, and lastly to $\delta'(2, c)$. Since there are already established valid symbol transitions at $\delta'(2, b)$ and $\delta'(2, c)$, there is no need to replace them. Then the transitions from the next state will be considered.

   At the end of iterating all transitions at state 2 there is no failure transition is provided from this state. Thus, $\mathfrak{f}(2)$ remains $\perp$. This is a case whereby a "useless" failure transition is not generated.

5. Transitions at state 3 are considered. The resulting transitions are shown in Tables 6.1 (i)-(j).

   The algorithm will now insert transitions starting at an entry for state 3 and symbol a. The transition $\delta'(3, a)$ currently contains an invalid entry, therefore the algorithm must insert some transitions. The maximum failure path length l is generated randomly. Let l to be 1, set h to be the current stat 3 and T is instantiated by adding state 3 into itself. A previously defined failure transition from state 3 to 2 is taken. The state 2 is added into T and it is assigned as the head h. Since a

single failure transition has been taken and because $\delta'(2, a)$ is a valid transition, namely 2, no further attempts to generate failure path generation are to be made. The transition value $\delta'(2, a) = 2$ is copied to the value at $\delta(2, a)$. As usual, for each state in T, the algorithm inserts two of DFA symbol transitions, i.e. $\delta'(2, a)$ and $\delta'(3, a)$ become 2.

The remaining two entries to look at for state 3 are the columns b and c, i.e. $\delta'(3, b)$ and $\delta'(3, c)$. They both have existing valid DFA symbol transitions namely: $\delta'(3, b) = 1$ and $\delta'(3, c) = 3$. Therefore, the algorithm will simply skip these last two transition iteretions.

At this stage, the two automata have been created: a complete DFA and a language equivalent FDFA. As expected, the complete DFA has 12 symbol transitions. The FDFA has 9 transitions, 3 being failure transitions and 6 being symbol transitions. Thus the FDFA shows a transition reduction of 3 — i.e. it has 25% fewer transitions than the DFA. This example was provided to facilitate an understanding the algorithm for the reader.

6.3 Measurements

In this investigation the structure of the random DFAs and corresponding random FDFAs have to the following properties.

- The states of the set Q are indicated by natural numbers starting from 0.
- In all cases, $|\Sigma| = 10, q_s = 0$, and F is a set of states randomly chosen in Q.
- $|Q|$ ranges over the interval [250, 500, 750, ..., 2250, 2500]. There are therefore 10 possible state set sizes.
- The maximum failure path size, k, allowed for the FDFAs ranges over the interval [10, 20, ..., 90, 100]. There are, therefore, 10 possible maximum failure path size settings.

- 100 pairs of random DFA and language equivalent FDFAs were generated using Algorithm 6.1[2] described above — one for each state size / maximum failure path size combination.

In this experiment, our concern is the transition reductions resulting from the FDFAs (i.e. DHA-FDFAs and D^2FA) that are derived from random DFAs. Specifically, we are interested in measuring the symbol transitions removed and the overall transition reduction. Here, only the data for percentage transition reductions by the FDFAs from the random DFAs will be provided. The flowchart in Figure 6.1 depicts the sequence of data measurement activities in this experiment.

Unlike in the case of having an "optimal" FDFA in the form of an AC-fail FDFA as described in the previous chapter, in the present case, no such optimal FDFA is available. However, Algorithm 6.1 provides at least one instance of a language-equivalent "randomly generated" FDFA of the original DFA. Consequently, we are primarily concerned with *comparing the number* of transitions of the generated FDFAs against this "randomly generated" FDFA, rather than with *comparing the placement* of transitions of the various FDFAs. This comparison relates to the failure transitions of the randomly generated FDFA and the failure transitions of other FDFAs. It also relates to symbol transitions between DFAs and their language-equivalent FDFAs.

Note that the D^2FA algorithm outputs will be included in the empirical results to be compared.

6.3.1 The Results

Preliminary investigation showed that the percentage reduction in symbol transitions in FDFAs remained largely unaffected by the size of the respective automata, i.e. by $|Q|$. However, the ratio of transitions reduction by the

[2]Available online at:
http://madodaspace.blogspot.co.za/2016/02/a-toolkit-for-failure-deterministic.html

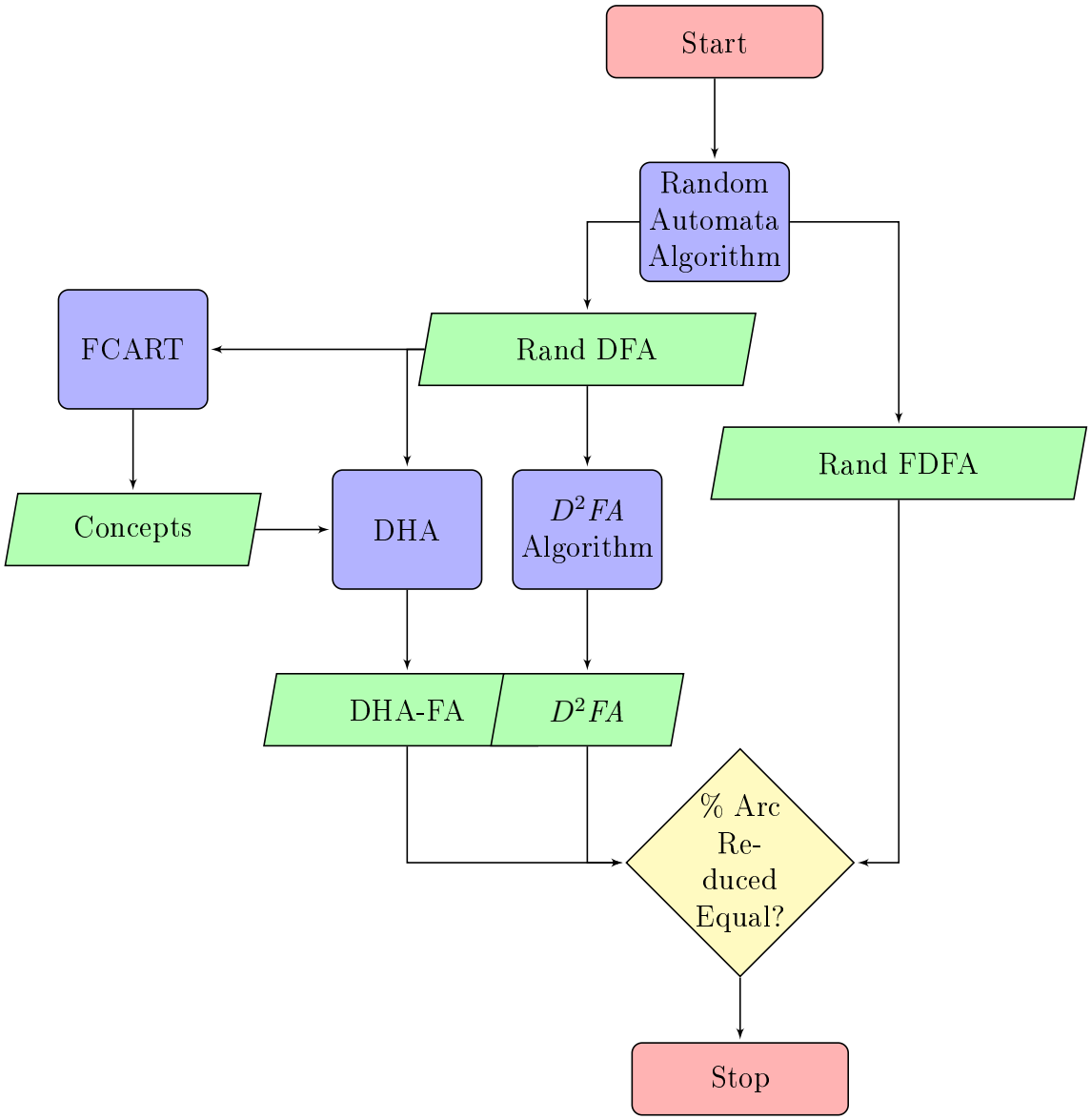

Figure 6.1: Comparing DHA-FDFAs and D^2FA against random FDFA for transition reduction from a random DFA.

FDFAs from random DFAs transition size is affected by the value of k. As a result of this observation, we will plot the transition reduction graphs by using k values as our $x - axis$ and the percentage transition savings as $y - axis$.

Hence, the initial data to be presented will be averages taken over all automata sizes (i.e averaged over $|Q| = [250, 500, \ldots 2500]$ for various values of k. As stated early in this section, the maximum failure path size has been altered for various FDFAs with $k = [10, 20, \ldots, 100]$. The percentage by which transitions are reduced may be expected to increase as k grows. As stated in Section 6.2.2, k is an input variable that is a maximum bounding value that can be assigned to the internal positive integer variable ℓ.

Symbol Transitions Removed

Figure 6.2 depicts the percentage of symbol transitions removed from the original DFA, on average, by each type of FDFA for various values of k. The percentage of symbol transition reduction in all linear plots in the figure (other than the graph for MaxAR) increases as k increases, up until when $k = 30$. Thereafter, for $k > 30$, a near-horizontal trend is observed.

The graph for D^2FA FDFA shows that this algorithm reduces the average number of symbol transitions in the DFA by more than the DHA heuristics. Indeed D^2FA algorithm also reduces transitions by more than the average reduction originally present in the randomly generated FDFA. The D^2FA FDFA's reached a maximum average of 85% for DFA symbol transitions reduction. In contrast, when $k > 30$, the DHA MaxIntent, MaxInt-MaxExt and DHA-MinExtent automata show near constant symbol transition reduction at just above 75%. The percentage of symbol transition reductions of the random FDFA grows more or less linearly when $k > 30$, and its graph is bounded above by the D^2FA graph and below by the three DHA-FDFAs graphs. In contrast to these trends, the MaxAR criterion's output achieved very low averages for percentages of symbol transition reductions. These values are below 40% for all points plotted.

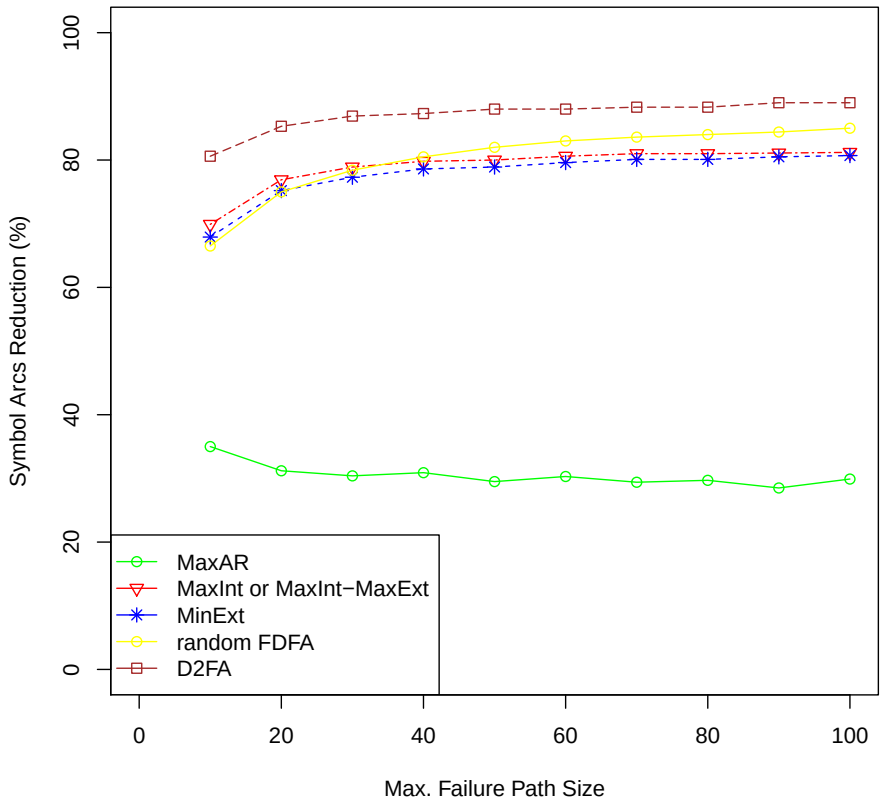

Figure 6.2: The average *symbol transitions* removed from the random DFA by the various FDFA types as a percentage of the transition size of random DFAs for different k values.

Overall Transition Reduction

The overall transition reductions brought about by the FDFAs over the random DFAs will now be briefly discussed. By overall transition reduction we mean the difference between the total number of DFA transitions (all of which are, of course, symbol transitions) and the total number of language-equivalent FDFA transitions (some if which are failure transitions and the rest, symbol transitions). This reduction is usually expressed as a percentage of DFA transitions. These percentage reductions, averaged over all DFA

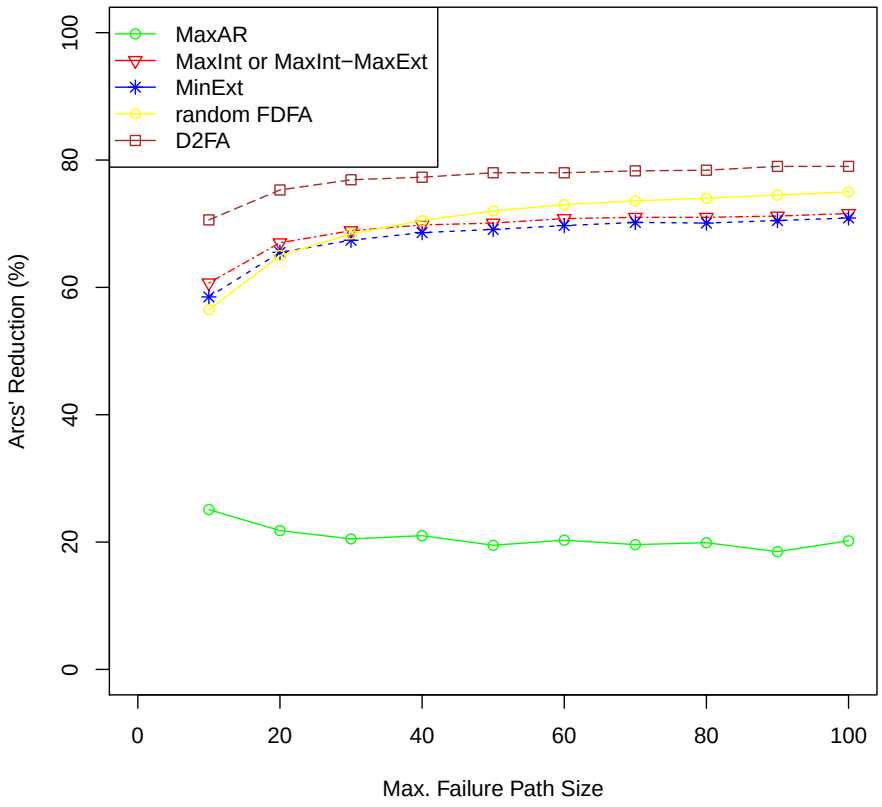

Figure 6.3: The average *overall transition* reduction by the various FDFA types as a percentage of the transition size of random DFAs for different k values.

sizes, is shown in Figure 6.3 for different k values. These graphs have the same shape as the graphs in Figure 6.2.

To give an insight into how the results are distributed over the various DFA sizes, Figure 6.4 depicts the same data, now no longer averaged over all DFA sizes, but in box-whisker plot format.

The DHA MaxIntent or MaxInt-MaxExt box-whisker plots show that there is minimal variation — most of the observations lie very close to their median values. The MinExt, random FDFAs and D^2FAs show slightly more

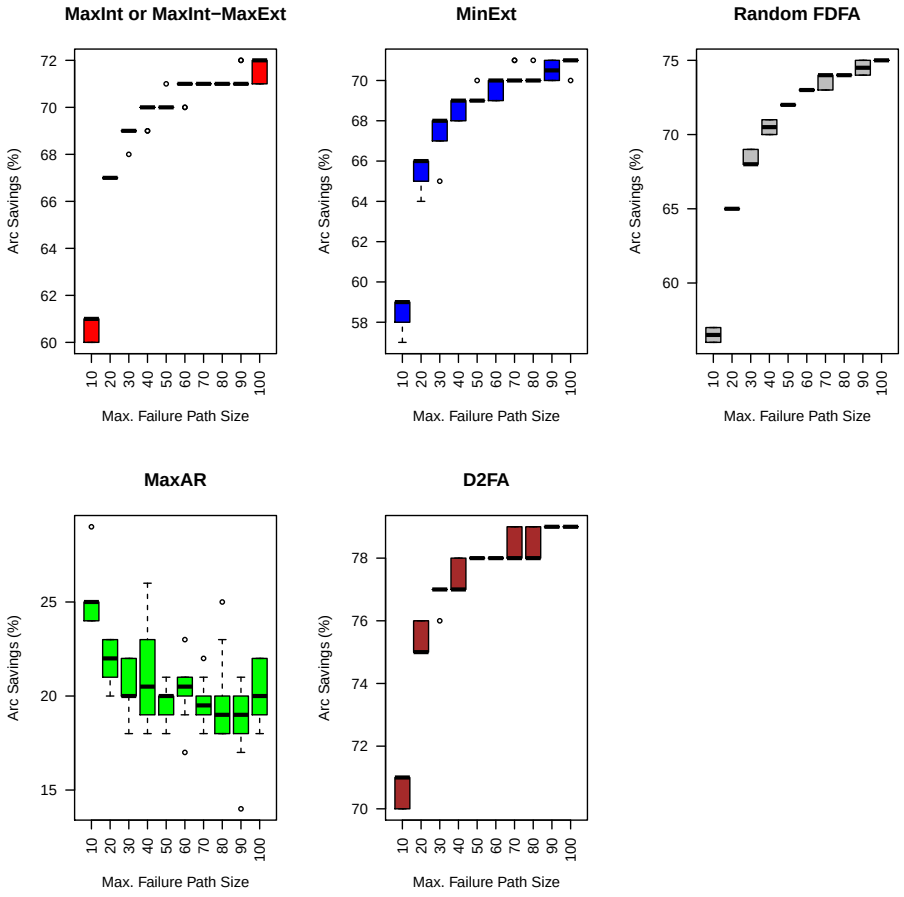

Figure 6.4: The boxplot graphs for the *overall transition* reduction by the various FDFA types as a percentage of the transition size of random DFAs for different k values.

variation, but the variation is still very modest. The MaxAR FDFAs show fairly large variations.

6.3.2 An Analysis of Results

The results presented above were with respect to five different ways of generating language-equivalent FDFAs from a DFA that was itself derived from a "randomly" constructed FDFA. This random FDFA was not specifically designed to be minimal (i.e. to be the smallest of all FDFAs that are language-

equivalent to the DFA in the sense of having the least number of transitions). It was therefore somewhat surprising to see that it generally had less transitions than the FDFAs produced by the DHA variants. That it was not minimal was illustrated by the fact that the D^2FA derived FDFAs had fewer transitions.

The overall transition reduction performance for the various language-equivalent FDFA may be expressed as follows:[3]

$$\begin{aligned} D^2FA &> random\_FDFA \\ &> MaxIntent\_FDFA \\ &= MaxInt\text{-}MaxExt\_FDFA \\ &> MinExtent\_FDFA \\ &> MaxAR\_FDFA \end{aligned}$$

The results showed that in no instance did a DHA FDFA variant obtain more transition reductions than the D^2FA algorithm.

It was interesting to note with respect to transition reductions that the three DHA heuristics, namely MaxIntent, MaxInt-MaxExt and MinExtent were able to produce FDFAs that were smaller than the random FDFAs, for as long as a maximum failure path size of 30 or less was used to generate the random FDFA. For larger values of k, these heuristics were unable to attain as many transition reductions as the original random FDFAs but they nevertheless worked reasonably well. However, the MaxAR heuristic remained the worst performer for generating FDFAs as close to the minimum as possible.

The linear graphs plots for D^2FA FDFA and DHA MaxIntent, MaxExtent and MinExtent FDFAs are parallel. This observation also shows that there is an approximate 10% difference between the graph for the FDFA produced by the D^2FA algorithm and the graphs for the three best performing DHA-

[3]The arithmetic operators are not used in their usual way but they are a notation for a ranking the FDFA algorithms based on transition reduction.

derived FDFAs. Given that $|\Sigma| = 10$ and the 10% transition difference between the three DHA variants and D^2FA then the difference in actual number of symbol transitions is estimated to be |Q|. The rationale for this observation stems from the following equation, where $(|\Sigma| \times |Q|)$ is the size of a DFA transitions.

$$10\% \; of \; (|\Sigma| \times |Q|) = \frac{10}{100} \times (10 \times |Q|) = |Q|$$

This suggests that the D^2FA algorithm out-performs the three DHA variants by an estimated difference of $|Q|$ symbol transitions.

The fact that there is a 10% performance difference between the best DHA heuristics and the D^2FA algorithm indicates suboptimal effects of the "greedy" approach used by the DHA variants. Whether this behaviour would be sustained for different values of $|\Sigma|$ and whether there are better heuristics that could be employed within the DHA context is left for future research.

The gradual increase of the random FDFA transition reductions is caused by the increase of the failure path sizes selected. With larger failure paths, chances of landing into existing symbol transitions become minimal — thus, reducing possibilities of generating new random symbol transitions.

6.4 Conclusion

The investigation was aimed at measuring the extent to which the DHA variants remove transitions from a DFA in the general case. The supposed general DFA to be used was a random DFA. Therefore, an algorithm was devised for generating a 'random' FDFA and its language-equivalent complete DFA. The random FDFA and D^2FA algorithms were included in comparing the extent to which FDFAs generated by DHA and the D^2FA algorithms produced the highest DFA transition reduction results.

Chapter 7

Conclusion and Future Work

The MaxAR strategy was the original heuristic proposed for the DFA approach [9]. It was used in initial tests because, based on theoretical considerations, it was thought that it would probably be a reasonably good heuristic for deciding where to replace symbol transitions with failure transition. This turned out not to be so. This lead to the development of the three heuristics (MaxIntent, MinExtent, Max Intent-MaxExtent) that modify DHA, and these turned out to lead to better transition reductions.

Generally, the empirical results provided a comparison of various FDFA types, and they were tested in different domains. They were firstly tested against the AC-fail FDFAs and later against the general case FDFAs. As a by-product of general case FDFA tests, an algorithm for generating a random FDFA and a language equivalent DFA was proposed. An alternate failure-DFA generating algorithm called D^2FA algorithm was also included in all experiments conducted. This algorithm had been missed by earlier FDFA research that formed the basis of the DHA approach because it was published in the network intrusion detection literature and was described in vocabulary that did not show up in early keyword searches.

The empirical results revealed that the modified DHA FDFAs bring about reasonably good transition reduction of minimal Aho-Corasick DFAs, pro-

ducing FDFAs that are very close to the AC-fail FDFAs. However, they also perform satisfactorily in the general context, though not quite as well as the D^2FA algorithm.

The relatively small alphabet size of 10 was dictated by unavoidable growth in the size of the associated concept lattices. Even though suitable strategies for trimming the lattice without losing important information (for example by not generating concepts with arc redundancy less than 2) are being investigated, it is recognised that use of DHA will always be constrained by the potential for the associated lattice to grow exponentially. Nevertheless, from a theoretical perspective a lattice-based DHA approach to FDFA generation is attractive because it encapsulates the solution space in which a minimal FDFA might be found — i.e. each ordering of its concepts maps to a possible language-equivalent FDFA that can be derived from a DFA and at least one such ordering will be a minimal FDFA.

The MaxAR heuristic, initially thought to probably be quite good, turned out to be not so effective in reducing transitions. Perhaps it could be used to derive FDFAs that lies in between the minimal FDFA and a DFA. Such FDFAs might have value in some applications in striking a balance between reducing space and processing time efficiency. This, of course, is a tradeoff to be made.

All the proposed variants of DHA were surpassed by the Random (F)DFA algorithm and D^2FA algorithm in the general case FDFA experiments. Whether different DHA heuristics can be discovered that will produce smaller FDFAs is a challenge for future research.

The D^2FA generation approach is not as constrained by space limitations as the DHA approach and in the present experiments it has performed reasonably well. A practical conclusion is to use the D^2FA since it is not only very effective in transition reduction, but it is also independent of concept lattices. In the original publication, a somewhat more refined version is reported that attempts to avoid unnecessary chains of failure transitions. Future research should examine the minimising potential of this refined version using gener-

alised DFAs as input and should explore more fully the relationship between these D^2FA based algorithms and the DHA algorithms.

Bibliography

[1] B. W. Watson, *Constructing Minimal Acyclic Deterministic Finite Automata*. Ph.D thesis, University of Pretoria, 2010.

[2] M. Crochemore and C. Hancart, "Automata for matching patterns," in *Handbook of Formal Languages* (S. A. Rozenberg G., ed.), pp. 1–48, Springer-Verlag, 1997.

[3] D. Revuz, "Minimisation of acyclic deterministic automata in linear time," *Theor. Comput. Sci.*, vol. 92, no. 1, pp. 181–189, 1992.

[4] J. A. Brzozowski, "Canonical regular expressions and minimal state graphs for definite events," *Mathematical Theory of Automata*, vol. 12, pp. 529–561, 1962.

[5] J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," technical report, STAN-CS-71-190, 1971. URL `http://i.stanford.edu/pub/cstr/reports/cs/tr/71/190/CS-TR-71-190.pdf` .

[6] J. Holub and J. Zdárek, eds., *Proceedings of the Prague Stringology Conference 2011, Prague, Czech Republic, August 29-31, 2011*, Prague Stringology Club, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2011.

[7] O. AitMous, F. Bassino, and C. Nicaud, "An efficient linear pseudo-minimization algorithm for aho-corasick automata," in *CPM* (J. Kärkkäinen and J. Stoye, eds.), vol. 7354 of *Lecture Notes in Com-*

puter Science, pp. 110–123, Springer, 2012.

[8] J. Bubenzer, "Minimization of acyclic DFAs," in Holub and Zdárek [6], pp. 132–146.

[9] D. G. Kourie, B. W. Watson, L. G. Cleophas, and F. Venter, "Failure deterministic finite automata," in *Proceedings of the Prague Stringology Conference 2012* (J. Holub and J. Zdárek, eds.), pp. 28–41, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2012.

[10] H. Björklund, J. Björklund, and N. Zechner, "Compact representation of finite automata with failure transitions," technical report, Umeä University, 2013. URL `http://www8.cs.umu.se/research/uminf/reports/2013/011/part1.pdf`.

[11] D. G. Kourie, B. W. Watson, L. Cleophas, T. Strauss, and F. Venter, "Failure deterministic finite automata. technical report," technical report, Faster Research Group, 2012. URL `http://www.fastar.org/publications/FASTAR-TR-2012.1.1%2020120410%20-%20FailureFAConstruction.pdf` .

[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. S. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Pisa, Italy, September 11-15, 2006* (L. Rizzo, T. E. Anderson, and N. McKeown, eds.), pp. 339–350, ACM, 2006.

[13] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[14] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *In IEEE Infocom, Hong Kong*, pp. 333–340, 2004.

[15] X. Zha and S. Sahni, "Highly compressed aho-corasick automata for efficient intrusion detection," in *ISCC*, pp. 298–303, 2008.

[16] M. E. Lesk and E. Schmidt, "Unix vol. ii," ch. Lex - A Lexical Analyzer Generator, pp. 375–387, Philadelphia, PA, USA: W. B. Saunders Company, 1990.

[17] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[18] S. Hasib, M. Motwani, and A. Saxena, "Importance of aho-corasick string matching algorithm in real world applications," in *IJCSIT*, pp. 467–469, 2013.

[19] L. Cleophas, "Towards SPARE time: A new taxonomy and toolkit of keyword pattern matching algorithms," masters thesis, Technische Universiteit Eindhoven, Aug 2003.

[20] J. Holub, "The prague stringology club," 2015. URL `http://www.stringology.org/` .

[21] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[22] D. G. Kourie and B. W. Watson, *The Correctness-by-Construction Approach to Programming*. Springer, 2012.

[23] B. W. Watson, *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, Sep 1995.

[24] H. Björklund, J. Björklund, and N. Zechner, "Compression of finite-state automata through failure transitions," *Theor. Comput. Sci.*, vol. 557, pp. 87–100, 2014.

[25] B. Ganter and R. Wille, *Formal concept analysis - mathematical foundations*. Springer, 1999.

[26] B. Ganter, G. Stumme, and R. Wille, eds., *Formal Concept Analysis, Foundations and Applications*, vol. 3626 of *Lecture Notes in Computer Science*, Springer, 2005.

[27] D. G. Kourie, B. W. Watson, F. Venter, and L. Cleophas, "Formal concept analysis applications in stringology," in *Festschrift for Bořivoj Melichar 2012* (B. W. W. Jan Holub and J. Žďárek, eds.), (Prague Stringology Club, Czech Technical University in Prague, Czech Republic), pp. 127–139, 2012.

[28] D. van der Merwe, S. A. Obiedkov, and D. G. Kourie, "Addintent: A new incremental algorithm for constructing concept lattices," in *ICFCA* (P. W. Eklund, ed.), vol. 2961 of *Lecture Notes in Computer Science*, pp. 372–385, Springer, 2004.

[29] B. Zhou, S. C. Hui, and K. Chang, "A formal concept analysis approach for web usage mining," in *Intelligent Information Processing II, IFIP TC12/WG12.3 International Conference on Intelligent Information Processing (IIP 2004), October 21-23, 2004, Beijing, China*, pp. 437–440, 2004.

[30] F. Venter, B. W. Watson, and D. G. Kourie, "Multiple keyword pattern matching using position encoded pattern lattices," in *Proceedings of The Ninth International Conference on Concept Lattices and Their Applications, Fuengirola (Málaga), Spain, October 11-14, 2012*, pp. 281–292, 2012.

[31] P. Linz, *An introduction to formal languages and automata (4. ed.)*. Jones and Bartlett Publishers, 2006.

[32] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3th ed., 2006.

[33] J. C. Martin, *Introduction to Languages and the Theory of Computation*. McGraw-Hill Higher Education, 4th ed., 2011.

[34] S. Dori and G. M. Landau, "Construction of aho corasick automaton in linear time for integer alphabets," in *CPM* (A. Apostolico, M. Crochemore, and K. Park, eds.), vol. 3537 of *Lecture Notes in Computer Science*, pp. 168–177, Springer, 2005.

[35] A. I. Jony, "Analysis of multiple string pattern matching algorithms," *International Journal of Advanced Computer Science and Information Technology (IJACSIT)*, vol. 3, no. 4, pp. 344–353, 2014.

[36] D. Pao, W. Lin, and B. Liu, "A memory-efficient pipelined implementation of the aho-corasick string-matching algorithm," *ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 2, pp. 10:1–10:27, 2010.

[37] L. Cleophas and B. W. Watson, "Taxonomy-based software construction of SPARE time: a case study," *Software, IEE Proceedings*, vol. 152, no. 1, pp. 29–37, 2005.

[38] D. E. Knuth, J. H. M. Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, 1977.

[39] L. Cleophas, D. G. Kourie, and B. W. Watson, "Weak factor automata: Comparing (failure) oracles and storacles," in *Proceedings of the Prague Stringology Conference 2013* (J. Holub and J. Žďárek, eds.), (Czech Technical University in Prague, Czech Republic), pp. 176–190, 2013.

[40] L. G. Cleophas, D. G. Kourie, and B. W. Watson, "Efficient representation of DNA data for pattern recognition using failure factor oracles," in *2013 South African Institute for Computer Scientists and Information Technologists, SAICSIT '13, East London, South Africa, October 7-9, 2013*, pp. 369–377, 2013.

[41] L. G. Cleophas, D. G. Kourie, and B. W. Watson, "Weak factor automata: the failure of failure factor oracles?," *South African Computer Journal*, vol. 53, pp. 1–14, 2014.

[42] P. Bille, I. L. Gørtz, and F. R. Skjoldjensen, "Subsequence automata

with default transitions," in *SOFSEM 2016: Theory and Practice of Computer Science - 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings*, pp. 208–216, 2016.

[43] M. E. Lesk and E. Schmidt, "Lex – a Lexical Analyzer Generator," tech. rep., Bell Laboratories, 1975. CS Technical Report No. 39.

[44] G. Klein, S. Rowe, J. Ouwens, and R. Decamps, "Jflex - the fast scanner generator for java," 2014. URL `www.jflex.de/manual.html` .

[45] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956.

[46] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. D. Pietro, "An improved DFA for fast regular expression matching," *Computer Communication Review*, vol. 38, no. 5, pp. 29–40, 2008.

[47] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[48] R. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[49] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, p. 345, 1962.

[50] A. Drozdek, *Data Structures and Algorithms in Java.* Delmar Learning, 3rd ed., 2008.

[51] M. Olivier, *Information Technology Research: A Practical Guide for Computer Science and Infomatics.* Van Schaik, 3rd ed., 2008.

[52] B. W. Watson and L. G. Cleophas, "SPARE parts: a c++ toolkit for string pattern recognition," *Softw., Pract. Exper.*, vol. 34, no. 7, pp. 697–710, 2004.

[53] L. Cleophas, B. W. Watson, J. Ouwens, and M. Frish-

ert, "SPARE time / SPARE parts 2003," 2003. URL http://fastar.org/main.php?button=spare_time .

[54] A. Buzmakov and A. Neznanov, "Practical computing with pattern structures in FCART environment," in *Proceedings of the International Workshop "What can FCA do for Artificial Intelligence?" (FCA4AI at IJCAI 2013), Beijing, China, August 5, 2013.*, pp. 49–56, 2013.

[55] A. Neznanov, D. Ilvovsky, and S. Kuznetsov, "FCART: A new FCA-based system for data analysis and knowledge discovery," in *Contributions to the 11th International Conference on Formal Concept Analysis*, pp. 65–78, 2013.

[56] A. Neznanov, D. Ilvovsky, and A. Parinov, "Advancing FCA workflow in FCART system for knowledge discovery in quantitative data," in *Proceedings of the Second International Conference on Information Technology and Quantitative Management, ITQM 2014, National Research University Higher School of Economics (HSE), Moscow, Russia, June 3-5, 2014*, pp. 201–210, 2014.

[57] A. Neznanov and A. Parinov, "FCA analyst session and data access tools in FCART," in *Artificial Intelligence: Methodology, Systems, and Applications - 16th International Conference, AIMSA 2014, Varna, Bulgaria, September 11-13, 2014. Proceedings*, pp. 214–221, 2014.

[58] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL http://www.R-project.org/ .

[59] T. Hothorn and B. S. Everitt, *A Handbook of Statistical Analyses Using R*. Chapman & Hall/CRC Press, 3rd ed., 2014.

[60] A. Neznanov and A. Parinov, "Analyzing social networks services using formal concept analysis research toolbox," in *Proceedings of the Workshop on Social Network Analysis using Formal Concept Analysis in conjunction with the 13th International Conference on Formal Concept Analysis (ICFCA 2015), Nerja (Malaga), Spain, June 25, 2015.*,

2015.

[61] K. J. Lang, "Random DFA's can be approximately learned from sparse uniform examples," in *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992.*, pp. 45–52, 1992.

[62] B. Balle, "Ergodicity of random walks on random dfa.," *CoRR*, vol. abs/1311.6830, 2013.

[63] R. Parekh and V. Honavar, "Learning DFA from simple examples," *Machine Learning*, vol. 44, no. 1/2, pp. 9–35, 2001.

[64] D. Berend and A. Kontorovich, "The state complexity of random DFAs," *CoRR*, vol. abs/1307.0720, 2013.

[65] D. Angluin and D. Chen, "Learning a random DFA from uniform strings and state information," in *Algorithmic Learning Theory - 26th International Conference, ALT 2015, Banff, AB, Canada, October 4-6, 2015, Proceedings*, pp. 119–133, 2015.

[66] C. Nicaud, "Random deterministic automata," in *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, pp. 5–23, 2014.

Appendix A

Acronyms

This appendix contains a list of *acronyms* that were used throughout the dissertation. They are shown in list below. The listed acronyms are alphabetically sorted and their meanings are placed alongside them.

Acronym	**Meaning**
AC	Aho-Corasick
AR	Arc Redundancy
ASCII	American Standard Code for Information Interchange
DFA	Deterministic Finite Automata
DHA	DFA - Homomorphic Algorithm
D^2FA	Delayed-input Deterministic Finite Automata
FA	Finite Automata
FCA	Formal Concept Analysis
FCART	Formal Concept Analysis Research Toolbox
FDFA	Failure Deterministic Finite Automata
FFO	Failure Factor Oracle
FO	Factor Oracle
KMP	Knuth-Morris-Pratt
$MaxAR$	Maximum Arc Redundancy

$MaxInt$	Maximum Intent
$MaxInt-MaxExt$	Maximum Intent - Maximum Extent
$MinExt$	Minimum Extent
NFA	Nondeterministic Finite Automata
PAR	Positive Arc Redundancy
RAM	Random Access Memory
XML	Extensible Markup Language
1D	1 Dimension
2D	2 Dimension

Appendix B

Symbols

This appendix contains a list of *symbols* used throughout the thesis. The listed acronyms are ordered by the chapters they first appeared in and their descriptions are placed alongside them.

Symbol	Description
Chapter 2	
Σ	an alphabet
$\|X\|$	number of elements in a set X
s	a string
ϵ	an empty string
Σ^*	The Kleene closure of Σ
Σ^+	$\Sigma^* - \{\epsilon\}$
$u.v$ - (dot)	concatenation of strings u and v
$\mathcal{D}$	a DFA
Q	a finite set of states of an automaton
δ	symbol transitions function mapping
F	a set of final states of an automaton
q_s	a start state of an automaton

$\perp$	an undefined state of an automaton
p	an element of Q (a state)
q	yet another state ($q \in Q$)
a	an element of an alphabet (i.e. $a \in \Sigma$)
$\rightarrow$	a total (complete) function
$\nrightarrow$	a partial function
$\mathcal{L}$	a language of an automaton
$\mathcal{F}$	an FDFA
$\mathfrak{f}$	failure transitions function mapping
$\overset{\mathfrak{f}}{\rightsquigarrow}$	a failure path
i	an index of a list (or set)
j	yet another index of a list
O	set of objects
A	set of attributes for the set of objects
I	a subset product of I and O i.e. $I \subseteq A \times O$
c	a formal concept
$ext(c)$	an extent of a concept c
$int(c)$	an intent of a concept c
P	a pattern set (a set of keywords)
N	number of keyword in a pattern set P
T	an input string to an automaton

Chapter 3

$\emptyset$	an empty set
O	variable in DHA for a set of states in Q (i.e $O \subseteq Q$)

Chapter 6

T	a variable that stores the elements in a failure path
k	a variable for maximum failure path size for an FDFA
l	a variable for max. failure path size from at a transition

Appendix C

Derived Publications

This appendix contains publications derived from the research done for this thesis.

- M. Nxumalo, D. Kourie, L. Cleophas, and B. Watson, "An Aho-Corasick based assessment of algorithms generating failure deterministic finite automata," in *Proceedings of the 12th International Conference on Concept Lattices and their Applications (CLA 2015)*, (Clermont-Ferrand, France), 2015.
- M. Nxumalo, D. Kourie, L. Cleophas, and B. Watson, "On Generating a Random Deterministic Finite Automaton as well as its Failure Equivalent", To appear in the proceedings of the *RuZA 2015 Workshop*, 2016.