FIRE Lite: FAs and REs in C++

Bruce W. Watson

Ribbit Software Systems Inc., IST Technologies Research Group
Box 24040, 297 Bernard Ave., Kelowna, B.C., V1Y 9P9, Canada
e-mail: `watson@RibbitSoft.com`, fax: +1 514 938 9308

Abstract. This paper describes a C++ finite automata toolkit known as FIRE Lite (FInite automata and Regular Expressions; Lite since it is the smaller and newer cousin of the FIRE Engine toolkit, also originally developed at the Eindhoven University of Technology). The client interface and aspects of the design and implementation are also described. The toolkit includes implementations of almost all of the known automata construction algorithms and many of the deterministic automata minimization algorithms. These implementations enabled us to collect performance data on these algorithms. The performance data, which we believe to be the first extensive benchmarks of the algorithms, are also summarized in this paper.

1 Introduction and related work

FIRE Lite is a C++ toolkit implementing finite automata and regular expression algorithms. The toolkit is a computing engine, providing classes and algorithms of a low enough level that they can be used in most applications requiring finite automata or regular expressions. Almost all of the algorithms derived in [Wat95, Chapter 6] are implemented. We begin by considering other toolkits.

1.1 Related toolkits

There are several existing finite automata toolkits. Some of the more closely related ones are described here:

- The FIRE Engine II system, as described at the Ribbit Software Systems Inc. web site, `www.RibbitSoft.com`. The FIRE Engine II is a commercial re-implementation of FIRE Lite. It includes a number of features not found in the FIRE Lite, such as: regular grammars (in addition to regular expressions), extended regular expressions, Mealy transducers (including their minimization), template parameterized transition labels of input and output alphabets, predicate transitions, dictionaries, and a full class library with multiple implementations providing different trade-offs with respect to running time and space. The performance of the implementation makes the toolkit suitable for use in natural language processing (involving automata with millions of states) and in compilers, to name a few applications. The FIRE Engine II is, however, much larger than FIRE Lite, making FIRE Lite more suitable for study by students of algorithmics, software engineering and programming.

- The FIRE Engine, as described in [Wat94a, Wat94b]. The FIRE Engine was the first of the toolkits from the Computing Science Faculty in Eindhoven. It is an implementation of all of the algorithms appearing in two early taxonomies of finite automata algorithms which appeared in [Wat93a, Wat93b]. The toolkit is somewhat larger than FIRE Lite (the FIRE Engine is 9000 lines of C++) and has a slightly larger and more complex public interface. The more complex interface means that the toolkit does not support multi-threaded use of a single finite automaton.
- The Grail system, as described in [RW93]. Grail follows in the tradition of such toolkits as Regpack [Leis77] and INR [John86], which were all developed at the University of Waterloo, Canada. It provides two interfaces:
  - A set of 'filter' programs (in the tradition of UNIX). Each filter implements an elementary operation on finite automata or regular expressions. Such operations include conversions from regular expressions to finite automata, minimization of finite automata, etc. The filters can be combined as a UNIX 'pipe' to create more complex operations; the use of pipes allows the user to examine the intermediate results of complex operations. This interface satisfies the first two (of three) aims of Grail [RW93]: to provide a vehicle for research into language theoretic algorithms, and to facilitate teaching of language theory.
  - A raw C++ class library provides a wide variety of language theoretic objects and algorithms for manipulating them. The class library is used directly in the implementation of the filter programs. This interface is intended to satisfy the third aim of Grail: an efficient system for use in application software.
- The Amore system, as described in [JPTW90]. The Amore package is an implementation of the semigroup approach to formal languages. It provides procedures for the manipulation of regular expressions, finite automata, and finite semigroups. The system supports a graphical user-interface on a variety of platforms, allowing the user to interactively and graphically manipulate the finite automata. The program is written (portably) in the C programming language, but does not provide a programmer's interface. The system is intended to serve two purposes: to support research into language theory, and to help explore the efficient implementation of algorithms solving language theoretic problems.
- The Automate system, as described in [CH91]. Automate is a package for the symbolic computation on finite automata, extended regular expressions (those with the intersection and complementation operators), and finite semigroups. The system provides a textual user-interface through which regular expressions and finite automata can be manipulated. A single finite automata construction algorithm (a variant of Thompson's) and a single deterministic finite automata minimization algorithm is provided (Hopcroft's). The system is intended for use in teaching and language theory research. The (monolithic) program is written (portably) in the C programming language, but provides no function library interface for programmers.

According to Pascal Caron (at the Université de Rouen, France), a new version of Automate is being written in the MAPLE symbolic computation system.

The provision of the C++ class interface in Grail makes it the only toolkit with aims similar to those of the FIRE Engine and of FIRE Lite. In the following section, we will highlight some of the advantages of FIRE Lite over the other toolkits.

1.2 Advantages and characteristics of FIRE Lite

The advantages to using FIRE Lite, and the similarities and differences between FIRE Lite and the existing toolkits are:

- FIRE Lite does not provide a user interface[1]. Some of the other toolkits provide user interfaces for the symbolic manipulation of finite automata and regular expressions. Since FIRE Lite is strictly a computing engine, it can be used as the implementation beneath a symbolic computation application.
- The toolkit is implemented for efficiency. Unlike the other toolkits, which are implemented with educational aims, it is intended that the implementations in FIRE Lite are efficient enough that they can be used in production quality software. This means that we have chosen to use compile-time constructs such as templates instead of inheritance and virtual methods. The use of templates and their methods can be optimized by the C++ compiler, whereas virtual function calls (and inheritance hierarchies) are inherently run-time constructs.
- Despite the emphasis on efficiency in FIRE Lite the toolkit still has educational value. The toolkit bridges the gap between the easily understood abstract algorithms appearing in [Wat95, Chapter 6] and practical implementation of such algorithms. The C++ implementations of the algorithms display a close resemblance to their abstract counterparts.
- Most of the toolkits implement only one of the known algorithms for constructing finite automata. For example, Automate implements only one of the known constructions. By contrast, FIRE Lite provides implementations of almost all of the known algorithms for constructing finite automata. (See [Wat95, Chapter 6] for a taxonomy of the algorithms.) Implementing many of the known algorithms has several advantages:
  - The client can choose between a variety of algorithms, given tradeoffs for finite automata construction time and input string processing time.
  - The efficiency of the algorithms (on a given application) can be compared.
  - The algorithms can be studied and compared by those interested in the inner workings of the algorithms.

[1] A rudimentary user interface is included for demonstration purposes.

2 Using the toolkit

In this section, we describe the client interface to the toolkit — including some examples which use the toolkit. The issues in the design of the current client interface are detailed in [Wat95].

There are two components to the client interface of FIRE Lite: regular expressions (class *RE*) and finite automata (classes whose names begin with *FA*...). We first consider regular expressions and their construction.

Regular expressions are implemented through class *RE*. This class provides a variety of constructors and member functions for constructing complex regular expressions. Stream insertion and extraction operators are also provided. (The public interface of *RE* is rather fat, consisting of a number of member functions intended for use by the constructors of finite automata.) The following program constructs a simple regular expression and prints it:

```
#include "re.hpp"
#include <iostream.h>

int main(void) {
    auto RE e( 'B' );
    auto RE f( CharRange( 'a', 'z' ) );
    e.concatenate( f );
    e.or( f );
    e.star();
    cout << e;
    return( 0 );
}
```

The header `re.hpp` defines regular expression class *RE*. The program first constructs two regular expressions. The first is the single symbol B. The second regular expression is a *CharRange* — a character range. The *RE* constructed corresponds to the range $[a, z]$ of characters. No particular ordering is assumed on the characters, though most platforms use the ASCII ordering. Character ranges can always be used as atomic regular expressions. The program concatenates regular expression f onto e and then unions f onto e. Finally, the Kleene closure operator is applied to regular expression e. The final regular expression, which is $((B \cdot [a, z]) \cup [a, z])^*$, is output (in a prefix notation) to standard output.

The abstract finite automata class defines the common interface for all finite automata; it is defined in `faabs.hpp`. A variety of concrete finite automata are provided in FIRE Lite; they are declared in header `fas.hpp`. There are two ways to use a finite automaton. In both of them the client constructs a finite automaton, using a regular expression as argument to the constructor. The two are outlined as follows:

1. In the simplest of the two, the client program calls finite automaton member function *FAAbs::attemptAccept*, passing it a string and a reference to an integer. The member function returns *TRUE* if the string was accepted by

the automaton, *FALSE* otherwise. Into the integer reference it places the index (into the string) of the symbol to the right of the last symbol processed.

2. In the more complex method, the client takes the following steps (which resemble the steps required in using a pattern matcher mentioned in [Wat95, Chapter 9]):
   (a) The client calls the finite automaton member function *FAAbs::reportAll*, passing it a string and a pointer to a function which takes an integer and returns an integer. As in the SPARE Parts (the keyword pattern matching toolkit which is a companion to FIRE Lite — see [Wat95, Chapter 9]), the function is the 'call-back' function.
   (b) The member function processes the input string. Each time the finite automaton enters a final state (while processing the string), the call-back function is called. The argument to the call is the index (into the input string) of the symbol immediately to the right of the symbol which took the automaton to the final state.
   (c) To continue processing the string, the call-back function returns *TRUE*, otherwise *FALSE*.
   (d) When the input string is exhausted, or the call-back function returns *FALSE*, or the automaton becomes stuck (unable to make a transition on the next input symbol), the member function *FAAbs::reportAll* returns the index of the symbol immediately to the right of the last symbol on which a successful transition was made.

The following program fragment takes a regular expression, constructs a finite automaton, and processes an input string:

```
#include "com-misc.hpp"
#include "re.hpp"
#include "fas.hpp"
#include <iostream.h>

static int report( int ind ) {
    cout << ind << '\n';
    return( TRUE );
}

void process( const RE& e ) {
    auto FARFA M( e );
    cout << M.reportAll( "hishershey", &report );
    return;
}
```

Header com-misc.hpp provides the definition of constants *TRUE* and *FALSE* (these will be eliminated with the new **bool** C++ datatype), while header fas.hpp gives the declarations of a number of concrete finite automata. Function *report* is used as the call-back function; it simply prints the index and returns *TRUE* to continue processing. Function process takes an *RE* and constructs a local finite automaton (of concrete class *FARFA*). It then uses the automaton processes string **hishershey**, writing the final index to standard output before returning.

Given these examples, we can now demonstrate a more complex use of a finite automaton. In this example, we implement a generalized Aho-Corasick pattern matcher (GAC — as in [Wat95, Chapter 5]) which performs regular expression pattern matching. Since regular expression pattern matching is not presently included in the SPARE Parts, this example illustrates how FIRE Lite could be used to implement such pattern matching for a future version of the SPARE Parts. (This example also highlights the fact that the call-back mechanism in FIRE Lite is very similar to the mechanism in the SPARE Parts.)

```
#include "re.hpp"
#include "fas.hpp"
#include "string.hpp"

class PMRE {
public:
    PMRE( const RE& e ) : M( e ) {}
    int match( const String& S, int cb( int ) ) {
        return( M.reportAll( S, cb ) );
    }
private:
    FARFA M;
};
```

Headers **re.hpp**, **fas.hpp**, and **string.hpp** have all been explained before. Class *PMRE* is the regular expression pattern matching class. Its client interface is modeled on the pattern matching client interfaces used in [Wat95, Chapter 9]. The class has a private finite automaton (in this case an *FARFA*) which is constructed from an *RE*. (Note that the constructor of class *PMRE* has an empty body, since the constructor of *FARFA M* does all of the work.) The member function *PMRE::match* takes an input string and a call-back function. It functions in the same way as the call-back mechanism in [Wat95, Chapter 9]. The member function is trivial to implement using member *FAAbs::reportAll*. Whenever the finite automaton enters an accepting state, a match has been found and it is reported.

An important feature of FIRE Lite (like SPARE Parts) is that the call-back client interface implicitly supports multi-threading. See [Wat95, Chapter 9] for a discussion of call-back functions and multi-threading.

3 The structure of FIRE Lite

In this section, we give an overview of the structure of FIRE Lite and some of the main classes in the toolkit.

In the construction algorithms of [Wat95, Chapter 6], the finite automata that are produced have states containing extra information. In particular, the 'canonical construction' produces automata whose states are dotted regular expressions, or items. Some of the other constructions produce automata with sets of items for states, or sets of positions for states, or derivatives for states, and so on.

The constructions of [Wat95, Chapter 6] appear as the constructors (taking a regular expression) of various concrete finite automata classes in FIRE Lite. It seems natural that mathematical concepts such as items, sets of items, positions, and sets of positions will also appear in such an implementation. The only potential problem is the performance overhead inherent in implementing automata with states which are sets of items, etc.

The solution used in FIRE Lite is to implement states with internal structure as *abstract-states* during the construction of a finite automaton. The finite automaton is constructed using the abstract-states (so that the constructor corresponds to one of the algorithms in [Wat95, Chapter 6]). Once the automaton is fully constructed, the abstract-states are too space and time consuming for use while processing a string (for acceptance by the automaton). Instead, we map the abstract-states (and the transition relations, etc) to *States* (which are simply integers) using a *StateAssoc* object. Once the mapping is complete, the abstract-states, their transition relations, and the *StateAssoc* object can be destroyed.

For all of the finite automata which are constructed from abstract-states, the constructor takes an initial abstract-state, which is used as a 'seed' for constructing the remaining states and the transition relation. For performance reasons, we wish to make the finite automaton constructor a template class whose template argument is the abstract-state class (this would avoid virtual function calls). Unfortunately, most compilers which are presently available do not support template member functions (which have recently been added to the draft C++ standard). As a result, we are forced to make the entire finite automata class into a template class. The main disadvantage is that this reduces the amount of object code sharing (see [Wat95, Chapter 8] for a discussion of the differences between source and object code sharing and templates versus inheritance).

Most of the abstract-state classes have constructors which take a regular expression. As a result, an *RE* can be used as argument to most of the finite automata classes — a temporary abstract-state will be constructed automatically.

There are three types of abstract-states, corresponding to finite automata with ε-transitions (see class *FAFA*), *ε-free* finite automata, and deterministic finite automata. Classes of a particular variety of abstract-state all share the same public interface by convention; the template instantiation phase of the compiler detects deviations from the common interface. For more on the three types of abstract-state, see [Wat95, Chapter 10].

The transition relations on *States* are implemented by classes *StateStateRel* (for ε-transitions), *TransRel* (for nondeterministic transitions), and *DTransRel* (for deterministic transitions).

The following table presents a summary of the various concrete automata classes and their template arguments (if any):

Class	*Description*	
FACanonical	Canonical automaton	
FAFA	Automaton template, with ε-transitions Single template argument required:	
		Abstract states
	ASItems	Item sets (canonical)
FARFA	ε-*free* automaton	
FAEFFA	Automaton template, without ε-transitions Single template argument required:	
		Abstract states
	ASEFItems	Items sets (no filter)
	ASEFPosnsBS	Berry-Sethi
	ASEFPosnsBSdual	dual of Berry-Sethi
	ASEFPDerivative	Antimirov
FADFA	Deterministic automaton template Single template argument required:	
		Abstract states
	ASDItems	Items sets (no filter)
	ASDItemsDeRemer	Items sets (DeRemer filter)
	ASDItemsWatson	Items sets ($\mathcal{W}$ filter)
	ASDPosnsMYG	McNaughton-Yamada-Glushkov
	ASDPosnsASU	Aho-Sethi-Ullman
	ASDDerivative	Brzozowski

4 The performance of FIRE Lite

In this section, we present performance data for the algorithms implemented in FIRE Lite. We begin by describing the testing methodology, followed by the performance of automata construction algorithms, and the time that each automaton type requires for a single transition; we conclude with the performance of the minimization algorithms.

4.1 Testing methodology

This section gives an overview of the methods used in gathering the test data. We begin with the details of the test environment, followed by the details of the methods used to generate the regular expressions for input to the algorithms.

Test environment All of the tests were performed on an IBM-compatible personal computer running MS-DOS. The machine has an INTEL PENTIUM processor with a 75 Mhz clock, an off-chip cache of 256 kilobytes and main memory of 8 megabytes. During all of the tests, no other programs which could consume processing power were installed.

The test programs were compiled with the WATCOM C++32 compiler (version 9.5a) with optimizations for speed. The WATCOM compiler is bundled with an MS-DOS extender (used to provide virtual memory for applications with large data-structures) known as DOS/4GW. Since the use of virtual memory could affect the performance data, all data-structures were made to fit in physical memory.

Timing was done by reprogramming the computer's built-in counter to count microseconds. This reprogramming was encapsulated within a C++ timer class which provided functionality such as starting and stopping the timer. Member functions of the class also subtracted the overhead of the reprogramming from any particular timing run.

Generating regular expressions A large number of regular expressions were randomly generated, from which the finite automata were built. We used the random number generator appearing in [PTVF92, p. 280]. The regular expressions were generated as follows:

1. A height in the range $[2, 5]$ was randomly chosen for the parse tree of the regular expression. (Larger heights were not chosen for memory reasons; smaller heights were not chosen since the constructions were performing close to the clock resolution.)
2. A regular expression of the desired height was generated, choosing between all of the eligible operators[2] with equal probability.
3. For the leaves, $\emptyset$ and ε nodes were never chosen. The $\emptyset$ was omitted, since such regular expressions prove to be uninteresting (they simply denote the empty language). Similarly, the ε was omitted, since the same effect is obtained by generating ? nodes[3].

There are a total of 3462 REs; the mean size is 9.63 nodes and the standard deviation is 4.72. The distribution of the number of nodes reflects the way in which the regular expressions were generated. Note that there are no regular expressions with a single node or with three nodes. These were omitted since the time to construct an FA from such small REs was usually below the resolution of the timer. (Two node regular expressions were used since they contain a $*$ or a $+$ node at the root. All of the constructions require more time to construct an automaton corresponding to such an expression.) Furthermore, it is not possible to generate lengthy strings in the language of such regular expressions.

Other statistics on the regular expressions were also collected, such as the height of the REs, the star-height of the REs, and some measure of the inher-

[2] Some operators will not be eligible. For example, to generate an RE of height 3, only the unary or binary operators can appear at the root.

[3] The ? operator is the regular operator used to indicate when something is optional.

ent nondeterminism in the REs[4]. These statistics did not prove to be useful in considering the performance of the algorithms.

Generating input strings For each type of automaton (FA, *ε-free* FA, and DFA), we also present data on the time required to make a single transition. In order to measure this, for each RE used as input to the constructions we generate a string in the prefix of the language denoted by the RE. Strings of length up to 10,000 symbols were generated.

The constructed automaton processes the string, making transitions, while the timer is used to measure the elapsed time. The time was divided by the number of symbols processed, yielding the average time for a single transition.

4.2 FA construction algorithm performance

In this section, we consider the time required (by each of the construction algorithms) to construct an automaton from a regular expression.

The algorithms The algorithms tested were derived in [Wat95, Chapter 6]. They have also been implemented in FIRE Lite. The implementations are discussed in detail in [Wat95]. For convenience, we will put the algorithms in two groups: those producing an FA, and those producing a DFA. The FA constructions are:

- The canonical construction (TH, since it is a variant of Thompson's construction), appearing in [Thom68] and [Wat95, Construction 6.15].
- The Berry-Sethi construction (BS), given in [BS86] and [Wat95, Construction 6.39].
- The dual of the Berry-Sethi construction (BS-D), appearing as [Wat95, Construction 6.65].

The DFA constructions are:

- The Aho-Sethi-Ullman construction (ASU) — [ASU86] and [Wat95, Construction 6.86].
- The deterministic item set construction (IC) — [Wat95, Example 6.23].
- DeRemer's construction (DER) — [Wat95, page 159].
- The filtered item set construction (FIC) — [Wat95, page 158].
- The McNaughton-Yamada-Glushkov construction (MYG), given as [MY60] and [Wat95, Construction 6.44].

For specific information on these algorithms, see [Wat95, Chapter 6].

Noticeably absent from this list are the derivatives-based algorithms (Brzozowski's and Antimirov's algorithms). The derivatives in these algorithms are

[4] One estimate of such nondeterminism is the ratio of alternation (union) nodes and $*$ or $+$ nodes to the total number of nodes in the regular expression.

used as the states. In their pure forms, the derivatives are stored as regular expressions. The space and time required to store and manipulate the regular expressions proved to be extremely costly, when compared to the representations of states used in some of the other constructions. The derivative-based algorithms consistently performed 5 to 10 times slower than the next slowest algorithm (the IC algorithm, in particular). Indeed, the preliminary testing could only be done for the smallest regular expressions without making use of virtual memory (which would further degrade their performance). No doubt the use of clever coding tricks would improve these algorithms greatly — though such coding tricks would yield a new algorithm.

Construction times For each of the generated regular expressions and each of the constructions, we measured the number of microseconds required to construct the automaton. For many applications, the construction time can be the single biggest factor in determining which construction to use.

First, the constructions are divided into two groups: those producing an FA (not a DFA), and those producing a DFA. The median performance of these two groups of constructions was graphed against the number of nodes in the regular expressions in Figures 1 and 2 respectively. All three of the algorithms

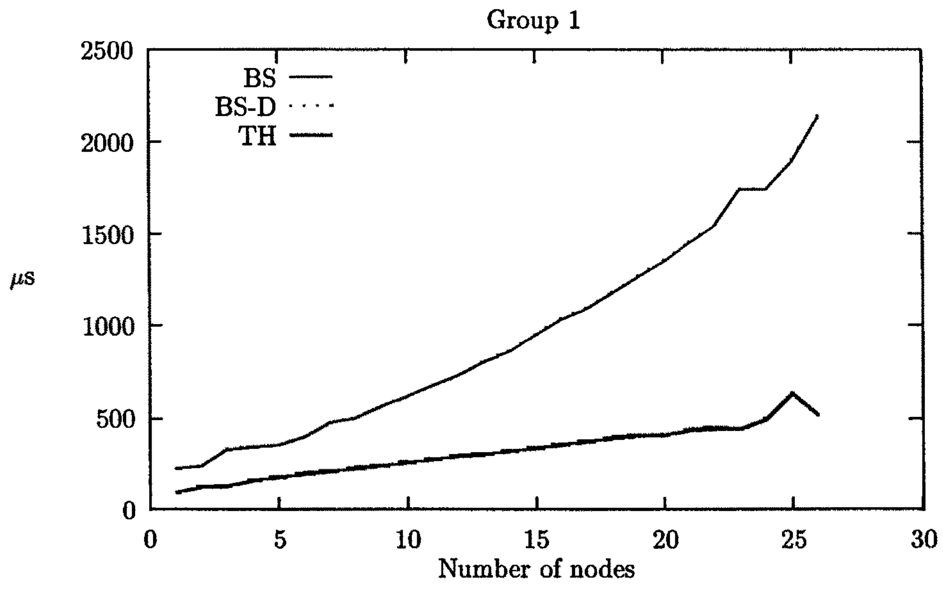

Fig. 1. Median construction times for FA constructions graphed against the number of nodes in the regular expressions. Note that BS and BS-D are superimposed as the higher ascending line.

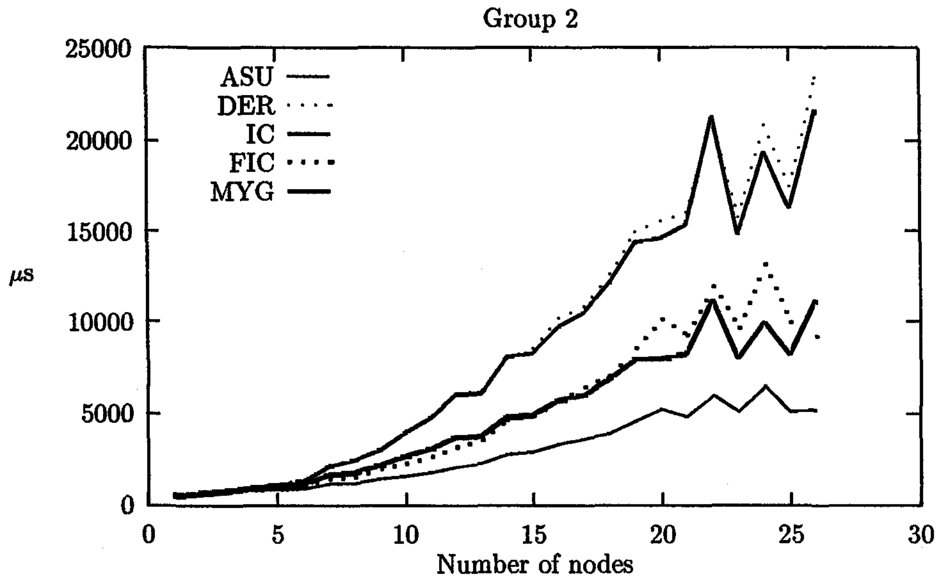

Fig. 2. Median construction times for DFA constructions graphed against the number of nodes in the regular expressions. The lowest line is ASU performance, while the middle pair of lines are MYG and FIC; the highest pair of lines is DER and IC.

in the first group are predicted to perform linearly in the number of nodes in the input regular expression. In Figure 1, the performance of the BS and BS-D constructions was nearly identical. They both performed somewhat worse than the TH algorithm. The apparent jump in construction time (of the TH algorithm) for 25 node regular expressions is due to the fact that only a single such expression was generated. Had more regular expressions been generated, the median performance would have appeared as a straight line (following the linear performance predicted for the TH algorithm).

The scale on Figure 2 shows that the second group of constructions was much slower than the first group. The ASU construction was by far the fastest, with FIC and MYG being the middle performers, and DER and IC being the slowest. In the range of 20 to 26 nodes, all of the constructions displayed peaks in their construction times. In these cases, some of the generated regular expressions have corresponding DFAs which are exponentially larger — forcing all of the constructions to take longer.

Constructed automaton sizes The size of each of the constructed automata was measured. The amount of memory space consumed by an automaton is directly proportional to the number of states in the automaton, and this data can be used to choose a construction based upon some memory constraints. Since the exact amount of memory (in bytes) consumed depends heavily on the compiler being used, we present the data in state terms.

Again, we group the non-DFA producing constructions and the DFA constructions. Figures 3 and 4 give the automata sizes versus number of nodes in the regular expressions for the two groups of constructions. The former figure

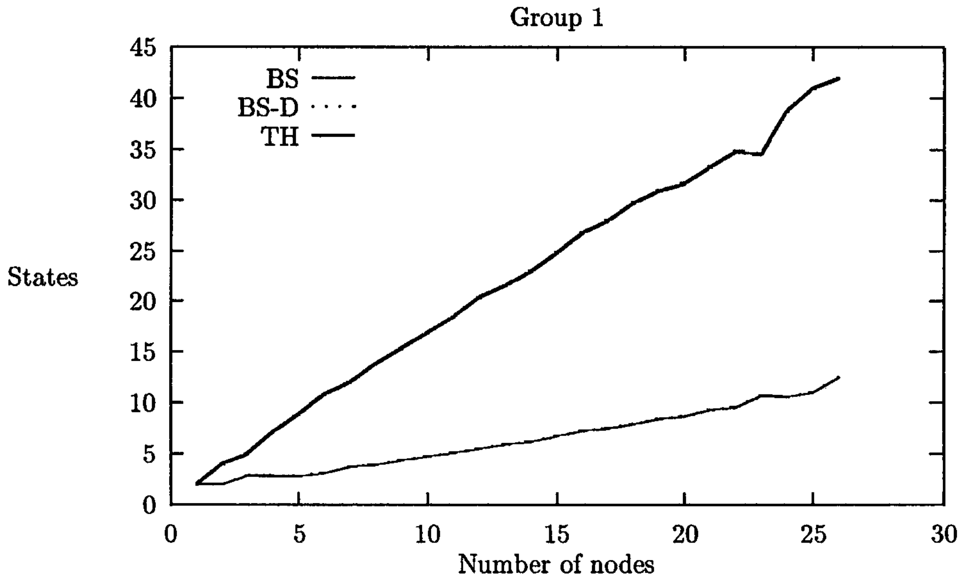

Fig. 3. The number of states in the FA is graphed against the number of nodes in the regular expression used as input for the TH, BS, and BS-D constructions. Note that the BS and BS-D constructions produce automata of identical size.

shows that the size of the TH-generated automata can grow quite rapidly. The BS and BS-D constructions produce automata of identical size In the second figure (Figure 4), we can identify two interesting properties of the constructions: ASU and FIC produce automata of the same size, as do the pair IC and MYG. Given the superior performance of ASU (over FIC), there is little reason to make use of FIC. Similarly, the MYG construction out-performs IC, and there is no reason to use IC since the automata will be the same size.

4.3 Automaton transition performance

The time required to make a single transition was measured for FAs (using TH), ε-*free* FAs (using BS), and DFAs (using FIC). The median time for the

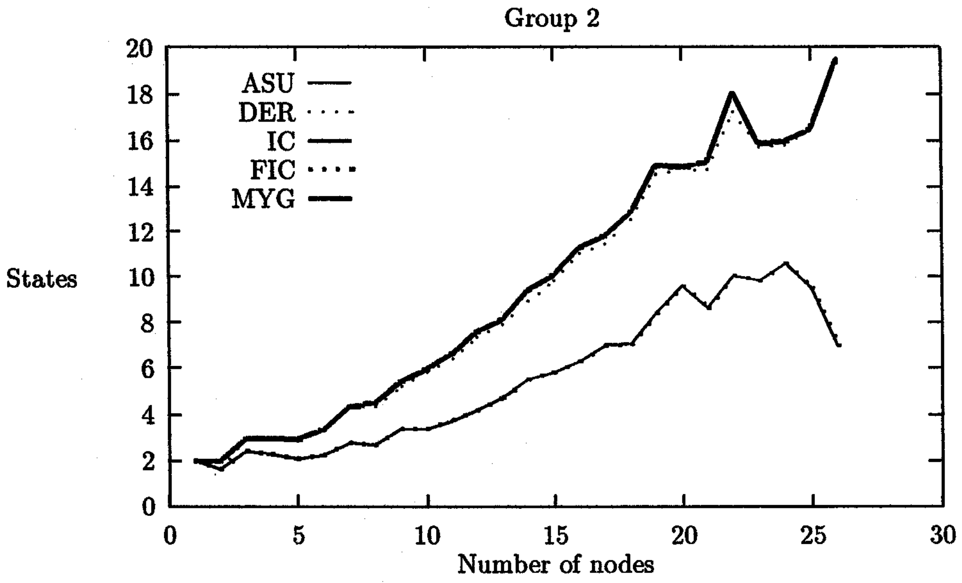

Fig. 4. The number of states in the DFA is graphed against the number of nodes in the regular expression used as input for the ASU, DER, FIC, IC, and MYG constructions. Note that the FIC and ASU constructions produce DFAs of identical size (the superimposed lower line), as did the pair IC and MYG (the superimposed higher line).

transitions has been graphed against the number of states in Figure 5. (Note that, for a given number of states, the number of each of the different types of automata varied.) The more general FAs displayed the slowest transition times, since the current set of states is stored as a set of integers, and the ε-transition and symbol transition relations are stored in a general manner. The *ε-free* FAs displayed much better performance, largely due to the time required to compute ε-transition closure in an automaton with ε-transitions. With the simple array lookup mechanism used in DFAs, it is not surprising that their transitions are by far the fastest (below one microsecond) and are largely independent of the number of states in the DFA.

The variance for general FAs and *ε-free* FAs is quite large. In both cases, the time for a single transition depends upon the number of states in the current state set. The variance for a DFA transition was also quite large. This is largely due to the fact that single transitions are around the resolution of the timer, meaning that other factors play a role. Such factors include artificial ones such as clock jitter and real ones such as instruction cache misses.

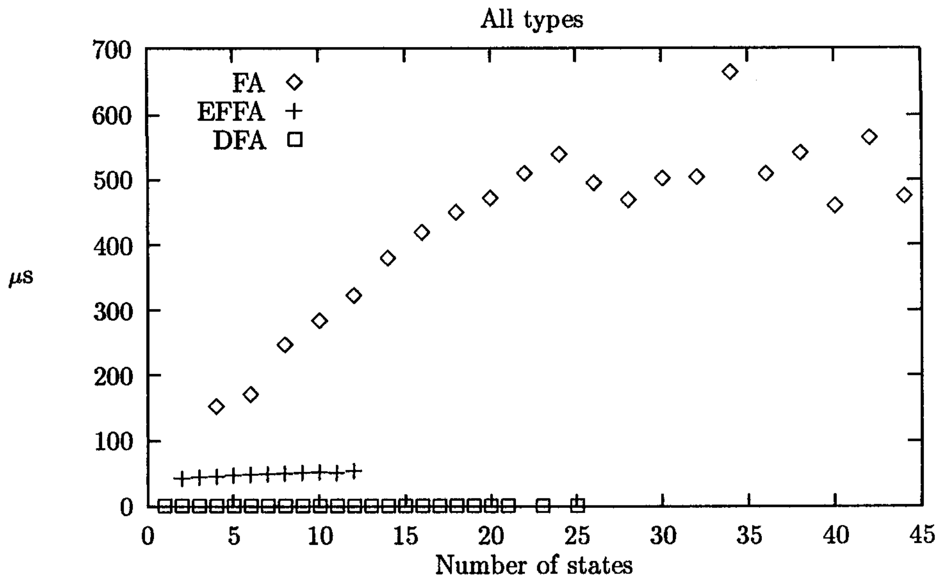

Fig. 5. Median time to make a single transition in microseconds (μs), for FAs, ε-*free* FAs, and DFAs, versus the number of states.

4.4 Automata construction algorithm recommendations

The conclusions relating to automata construction algorithms are:

- The advantages of the 'filters' introduced in [Wat95, Chapter 6] can be seen in the performance of IC, which is worse than either DER or FIC. Furthermore, IC produces larger automata than either DER or FIC.
- Predictably, the subset construction (with start-unreachable state removal) is a costly operation. All of the DFA construction algorithms were significantly slower than the general FA constructions.
- ASU is the subset construction composed with BS-D, while MYG is the subset construction composed with BS. Given the identical performance of BS and BS-D, it is interesting to note that ASU was significantly faster than MYG. Furthermore, ASU produced smaller DFAs than MYG.
- The time to make a transition can vary widely for different types of automata. As such, it can be an important factor in choosing a type of automaton.
  - Both FAs with ε-transitions and ε-*free* FAs have transition times that depend upon the number of states in the automaton; however, FAs with ε-transitions have significantly longer transition times.

- DFAs have transition times which are largely independent of the size of the automaton[5]. These times were around the resolution of the clock[6].

4.5 DFA minimization algorithms

Very little is known about the performance of DFA minimization algorithms in practice. Most software engineers choose an algorithm by reading their favourite text-book, or by attempting to understand Hopcroft's algorithm — the best known algorithm, with $\mathcal{O}(n \log n)$ running time. The data in this section will show that somewhat more is involved in choosing the right minimization algorithm. In particular, the algorithms appearing in a popular formal languages text-book [HU79] and in a compiler text-book [ASU86] have relatively poor performance (for the chosen input data). Two of the algorithms which could be expected to have poor performance[7] actually gave impressive results in practice. Recommendations for software engineers will be also be given.

The algorithms The five algorithms are fully derived and taxonomized in [Wat95, Chapter 7] and their corresponding implementations are described in detail in [Wat95, Chapter 11]. Here, we give a brief summary of each of the algorithms (and an acronym which will be used in these sections to refer to the algorithm):

- The Brzozowski algorithm (BRZ), first derived in [Brzo62]. It is unique in being able to process a FA (not necessarily a DFA), yielding a minimal DFA which accepts the same language as the original FA.
- The Aho-Sethi-Ullman algorithm (ASU), appearing in [ASU86]. It computes an equivalence relation on states which indicate which states are indistinguishable. The appearance of this algorithm in [ASU86] has made it one of the most popular algorithms among implementors.
- The Hopcroft-Ullman algorithm (HU), appearing in [HU79]. It computes the complement of the relation computed by the Aho-Sethi-Ullman algorithm. Since this algorithm traverses transitions in the DFA in the reverse direction, it is at a speed disadvantage in most DFA implementations (including the one used in FIRE Lite and the FIRE Engine).
- The Hopcroft algorithm (HOP), presented in [Hopc71]. This is the best known algorithm (in terms of running time analysis) with running time of $\mathcal{O}(n \log n)$ (where n is the number of states in the DFA).

[5] Although inspecting the implementation reveals that very dense transition graphs will yield more costly transitions than a sparse transition graph.

[6] This does not invalidate the results, since the average transition time is taken over a large number of transitions.

[7] They are Brzozowski's algorithm (which uses the costly subset construction) and a new algorithm which has exponential worst-case running time.

- The new algorithm (BW) appearing as [Wat95, Algorithm 7.28]. It computes the equivalence relation (on states) from below (with respect to the refinement ordering). The practical importance of this is explained in [Wat95, Chapter 7].

These algorithms are presently the only ones implemented in FIRE Lite.

Results For each of the random DFAs and each of the five algorithms, we measured the time in microseconds (μs) required to construct the equivalent (accepting the same language) minimal DFA.

The performance of the algorithms was graphed against the number of states in the original DFA. Graphing the performance against the number of edges in the original DFA was not found to be useful in evaluating the performance of the algorithms. The algorithms can be placed in two groups, based upon their performance. In order to aid in the comparison of the algorithms, we present graphs for these two groups separately.

The first group (ASU and HU) are the slowest algorithms; the graph appears in Figure 6. The HU algorithm was the worst performer of the five algorithms. It traverses the transitions (in the input DFA) in the reverse direction. A typical implementation of a DFA does not favour this direction of traversal. The ASU algorithm performed slightly better, although its performance is also far slower than any of BRZ, HOP, or BW. The second group (BRZ, HOP, and BW) is significantly faster; the corresponding graph appears in Figure 7. Note that this graph uses a different scale from the one in Figure 6. The data point (for 17 states) for the BW algorithm was dropped, since (at that point) the algorithm was more than 30 times slower than any of the other algorithms in this group (this is in keeping with the exponential worst-case running time of the algorithm).

4.6 Minimization algorithm conclusions and recommendations

We can draw the following conclusions from the data presented in these sections:

- Given their relative performance, the five algorithms can be put into two groups: the first consisting of ASU and HU, and the second consisting of BRZ, HOP, and BW.
- The HU algorithm has the lowest performance of all of the algorithms. This is largely due to the fact that it traverses the transitions of the DFA in the reverse direction — a direction not favoured by most practical implementations of DFAs.
- The ASU algorithm also displays rather poor performance. Traditionally, the algorithm has been of interest because it is easy to understand. The simplicity of BRZ minimization algorithm makes it even more suitable for teaching purposes.
- The HOP algorithm is the best known algorithm (in term of theoretical running time), with $\mathcal{O}(n \log n)$ running time. Despite this, it is the worst of

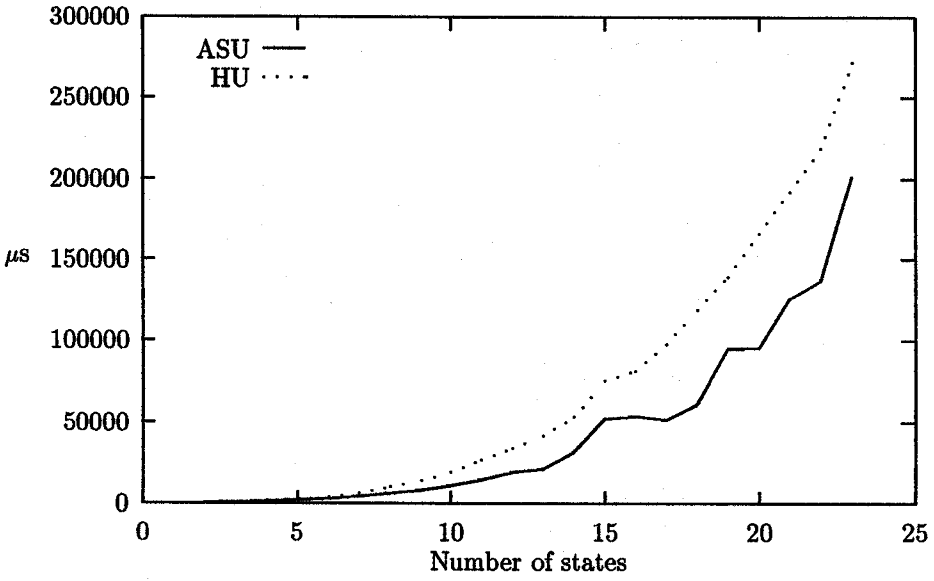

Fig. 6. Median performance (in microseconds to minimize) versus DFA size.

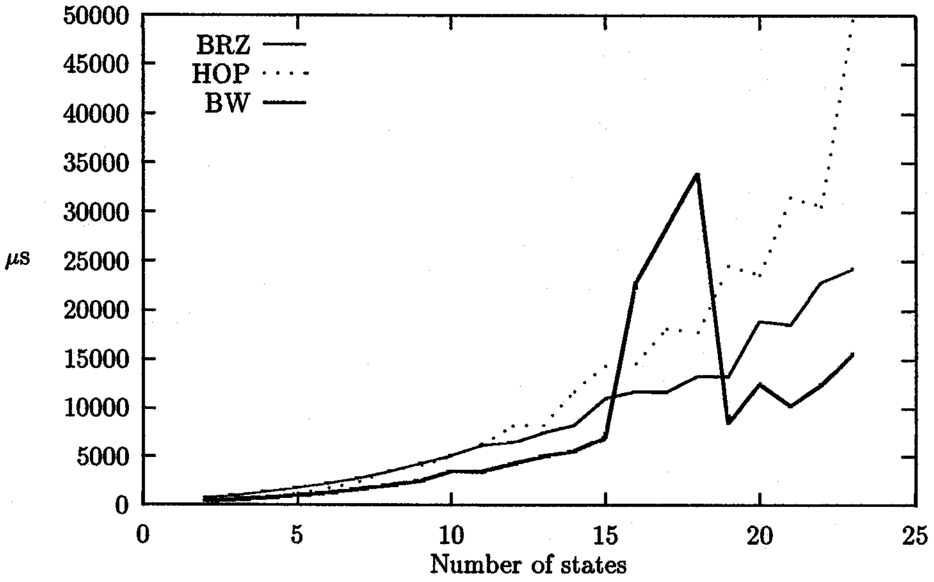

Fig. 7. Median performance (in microseconds to minimize) versus DFA size.

the second group of algorithms. With its excellent theoretical running time, it will outperform the BRZ and BW algorithms on extremely large DFAs. With memory constraints, we were unable to identify where the crossing-point of their performance is.
- The BRZ algorithm is extremely fast in practice, consistently outperforming Hopcroft's algorithm (HOP). This result is surprising, given the simplicity of the algorithm. The implementation of this algorithm constructs an intermediate DFA; the performance can be further improved by eliminating this intermediate step.
- The new algorithm (BW) displayed excellent performance. The algorithm is frequently faster than even Brzozowski's algorithm. Unfortunately, this algorithm can be erratic at times — not surprising given its exponential running time. The algorithm can be further improved using memoization.

Given these conclusions, we can make the following recommendations:

1. Use the new algorithm (BW, appearing as [Wat95, Algorithm 7.28]), especially in real-time applications (see [Wat95, Chapter 7] for an explanation of why this algorithm is useful for real-time applications). If the performance is still insufficient, modify the algorithm to make greater use of memoization.
2. Use Brzozowski's algorithm (derived in [Brzo62] and [Wat95, Chapter 7]), especially when simplicity of implementation or consistent performance is desired. The algorithm is able to deal with an FA as input (instead of only DFAs), producing the minimal equivalent DFA. When a minimization algorithm is being combined with a FA construction algorithm, Brzozowski's minimization algorithm is usually the best choice. The DFA construction algorithms are usually significantly slower than the FA construction algorithms, as was shown in the previous sections. For this reason, a FA construction algorithm combined with Brzozowski's minimization algorithm will produce the minimal DFA faster than a DFA construction algorithm combined with any of the other minimization algorithms.

   Brzozowski's algorithm can be further improved by eliminating the DFA which is constructed in an intermediate step.
3. Use Hopcroft's algorithm for massive DFAs. It is not clear from the data in this paper precisely when this algorithm becomes more attractive than the new one or Brzozowski's.
4. The two most-commonly taught text-book algorithms (the Aho-Sethi-Ullman algorithm and the Hopcroft-Ullman algorithm) do not appear to be good choices for high performance. Even for simplicity of implementation, Brzozowski's algorithm is better.

5 Future directions for the toolkit

A number of improvements to FIRE Lite will appear in future versions:

- Presently, FIRE Lite implements only acceptors. Transducers, such as Moore and Mealy machines (as would be required for some types of pattern matching, lexical analysis, and communicating finite automata), will be implemented in FIRE Lite and will be described at the workshop.
- A future version of the toolkit will include support for extended regular expressions, i.e. regular expressions containing intersection or complementation operators, and for linear (regular) grammars.
- Basic regular expressions and automata transition labels are represented by character ranges. A future version of FIRE Lite will permit basic regular expressions and transition labels to be built from more complex data-structures. For example, it will be possible to process a string (vector) of structures.

The future directions for the toolkit are presented in more detail in [Wat96] and at the Ribbit Software Systems Inc. web site, `www.RibbitSoft.com`.

6 FIRE Lite experiences and conclusions

A large number of people (worldwide) have made use of the FIRE Engine, and a number of people have started using FIRE Lite. As a result, a great deal of experience and feedback has been gained with the use of the finite automata toolkits. Some of these are listed here.

- The FIRE Engine and FIRE Lite toolkits were both created before the SPARE Parts. Without experience writing class libraries, it was difficult to devise a general purpose toolkit without having a good idea of what potential users would use the toolkit for. FIRE Lite has evolved to a form which now resembles the SPARE Parts (for example, the use of call-back functions).
- The FIRE Engine interface proved to be general enough to find use in the following areas: compiler construction, hardware modeling, and computational biology. The additional flexibility introduced with the FIRE Lite (the call-back interface and multi-threading) promises to make FIRE Lite even more widely applicable.
- Thanks to the documentation and structure of the FIRE Engine and FIRE Lite, they have both been useful study materials for students of introductory automata courses.
- The SPARE Parts was developed some two years after the taxonomy in [Wat95, Chapter 4] had been completed. By contrast, FIRE Lite was constructed concurrently with the taxonomy presented in [Wat95, Chapter 6]. As a result, the design phase was considerably more difficult (than for the SPARE Parts) without a solid and complete foundation of abstract algorithms.
- After maintaining and modifying several upgrades of the FIRE Engine, FIRE Lite is likely to be considerably easier to maintain and enhance.

Acknowledgements: I would like to thank Darrell Raymond, Derick Wood, Richard Watson, Nanette Saes, and Caroline Sori for their assistance in preparing this paper.

References

[ASU86] AHO, A.V., R. SETHI, and J.D. ULLMAN. *Compilers: Principles, Techniques, and Tools.* (Addison-Wesley, Reading, MA, 1988).

[Brzo62] BRZOZOWSKI, J.A. Canonical regular expressions and minimal state graphs for definite events, in: *Mathematical theory of Automata, Vol. 12 of MRI Symposia Series.* (Polytechnic Press, Polytechnic Institute of Brooklyn, NY, 1962) 529–561.

[BS86] BERRY, G. and R. SETHI. From regular expressions to deterministic automata, *Theoretical Computer Science* **48** (1986) 117–126.

[CH91] CHAMPARNAUD, J.M. and G. HANSEL. Automate: A computing package for automata and finite semigroups, *J. Symbolic Computation* **12** (1991) 197–220.

[Hopc71] HOPCROFT, J.E. An $n \log n$ algorithm for minimizing the states in a finite automaton, in: Z. Kohavi, ed., *The Theory of Machines and Computations.* (Academic Press, New York, 1971) 189–196.

[HU79] HOPCROFT, J.E. and J.D. ULLMAN. *Introduction to Automata, Theory, Languages, and Computation.* (Addison-Wesley, Reading, MA, 1979).

[John86] JOHNSON, J.H. INR: A program for computing finite automata, Technical Report, Department of Computer Science, University of Waterloo, Waterloo, Canada, January 1986.

[JPTW90] JANSEN, V., A. POTHOFF, W. THOMAS, and U. WERMUTH. A short guide to the Amore system, *Aachener Informatik-Berichte* **90**(02), Lehrstuhl für Informatik II, (Universität Aachen, January 1990).

[Leis77] LEISS, E. Regpack: An interactive package for regular languages and finite automata, Research Report CS-77-32, Department of Computer Science, University of Waterloo, Waterloo, Canada, October 1977.

[MY60] MCNAUGHTON, R. and H. YAMADA. Regular expressions and state graphs for automata, *IEEE Trans. on Electronic Computers* **9**(1) (1960) 39–47.

[PTVF92] PRESS, W.H., S.A. TEUKOLSKY, W.T. VETTERLING and B.P. FLANNERY. *Numerical Recipes in C: The Art of Scientific Computing.* (Cambridge University Press, Cambridge, England, 2nd edition, 1992).

[RW93] RAYMOND, D.R. and D. WOOD. The Grail papers, Department of Computer Science, University of Waterloo, Canada. Available by ftp from `CS-archive.uwaterloo.ca`.

[Thom68] THOMPSON, K. Regular expression search algorithms, *Comm. ACM* **11**(6) (1968) 419–422.

[Wat93a] WATSON, B.W. A taxonomy of finite automata construction algorithms, Computing Science Report 93/43, Eindhoven University of Technology, The Netherlands, 1993. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/automata/`.

[Wat93b] WATSON, B.W. A taxonomy of finite automata minimization algorithms, Computing Science Report 93/44, Eindhoven University of Technology, The Netherlands, 1993. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/automata/`.

[Wat94a] WATSON, B.W. An introduction to the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions, Computing Science Report 94/21, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/automata/`.

[Wat94b] WATSON, B.W. The design and implementation of the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions, Computing Science Report 94/22, Eindhoven University of Technology, The Netherlands, 1994. Available for ftp from `ftp.win.tue.nl` in directory `/pub/techreports/pi/automata/`.

[Wat95] WATSON, B.W. *Taxonomies and Toolkits of Regular Language Algorithms.* (Eindhoven University of Technology, The Netherlands, ISBN 90-386-0396-7, 1995).

[Wat96] WATSON, B.W. Implementing and using finite automata toolkits, in: Wahlster, W., *Proceedings of the 12th European Conference on Artificial Intelligence* (Budapest, Hungary, August 1996).