Improving Automata Efficiency by Stretching and Jamming

Noud de Beijer[1], Loek Cleophas[1,2], Derrick G. Kourie[1], and Bruce W. Watson[1⋆]

[1] FASTAR Research Group, Department of Computer Science, University of Pretoria, 0002 Pretoria, Republic of South Africa, http://www.fastar.org

[2] Software Engineering & Technology Group, Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, http://www.win.tue.nl/set

nouddebeijer@gmail.com, loek@loekcleophas.com, dkourie@cs.up.ac.za, bruce@fastar.org

Abstract. In recent years, the range of alphabet sizes typically used in applications of finite automata has grown considerably, now ranging from DNA alphabets—whose symbols are representable using 2 bits—to Unicode alphabets—whose symbol representation may take up to 32 bits. As automata traditionally use symbol encodings taking 8 bits, the different alphabet and symbol sizes bring up the question whether they may be exploited to either decrease memory use for the automata's transition tables or to decrease string processing time. In [3], *stretching* and *jamming* were introduced as transformations on finite automata. Given a finite automaton, we can stretch it by splitting each single transition into two or more sequential transitions, thereby introducing additional intermediate states. Jamming is the inverse transformation, in which two or more successive transitions are joined into a single transition, thereby removing redundant intermediate states. In this paper, we only consider a restricted form of stretching and jamming, in which a fixed factor is used to stretch (jam) transitions (transition paths) in a given automaton, and in which transition symbols are assumed to be encoded as bit strings. We consider improved versions of the algorithms that were presented in [3] for this particular form of stretching and jamming. The algorithms were implemented in C++ and used to benchmark the transformations. The results of this benchmarking indicate that, under certain conditions, stretching may be beneficial to memory use to the detriment of processing time, while jamming may be beneficial to processing time to the detriment of memory use. The latter seems potentially useful in the case of DNA processing, while the former may be for Unicode processing.

Keywords: finite automata; transformation; split transition; join transition; transition table size; string processing time

1 Introduction

In recent years, the range of alphabet sizes typically used in applications of finite automata has grown considerably. The alphabet typically used to be restricted to (some subset of) the 256 symbol ASCII set or a similarly sized alphabet. Nowadays the prevalence of data processing in bioinformatics and the frequent use of internationalisation in text processing have lead to the frequent use of much smaller alphabets for DNA, RNA and proteins on the one hand and of much larger alphabets for various encodings of Unicode on the other hand. Such alphabets require quite different numbers of bits to encode, ranging from 2 bits for DNA to up to 32 bits for certain encodings of Unicode. As automata representations typically used symbols of 8

⋆ This author is now also with the Department of Information Science, Stellenbosch University, Private Bag X1, 7602 Matieland, Republic of South Africa, http://www.informatics.sun.ac.za

Noud de Beijer, Loek Cleophas, Derrick G. Kourie, Bruce W. Watson: Improving Automata Efficiency by Stretching and Jamming, pp. 9–24 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8 © Czech Technical University in Prague, Czech Republic

bits, this begs the question whether it makes sense to consider multiple automata transitions on symbols from e.g. a 2 bit alphabet as a single transition for a new, larger 8 bit alphabet, or a single transition on a symbol from e.g. a 32 bit alphabet as multiple transitions from a smaller 8 bit alphabet. The aim of such transformations would be to either decrease memory use for the transition tables or to decrease string processing time.

In [3], stretching and jamming were introduced as transformations on finite automata. Given a finite automaton, we can stretch it by splitting each single transition into two or more sequential transitions, thereby introducing additional intermediate states while decreasing the size of the alphabet. Jamming is the inverse transformation, in which two or more successive transitions are joined into a single transition, thereby removing redundant intermediate states yet increasing the size of the alphabet. Here, we mainly consider a restricted form of stretching and jamming, in which a fixed factor is used to stretch (jam) every transition in a given automaton, and in which transition symbols are assumed to be encoded as bit strings. We consider slightly improved versions of the algorithms presented for this type of stretching and jamming in [3]. The improved version of the stretching algorithm does not introduce (additional) nondeterminism during stretching. For jamming, we consider situations in which the original algorithm is not applicable and discuss a number of solutions for this situation. The algorithms were implemented in C++ and used to benchmark the transformations. Our benchmarking results indicate that, under certain conditions, stretching may be beneficial for memory use to the detriment of processing time, while jamming may be beneficial for processing time to the detriment of memory use.

As far as we are aware, with the exception of the earlier paper by de Beijer et al [3], no work on this topic has been published so far, although some work on the upper bound on the number of states [5] has been done.

2 Preliminaries

In this section we present the basic notions used in this paper. Most of the notation used is standard (see for example [4]) but some new notation is introduced.

A deterministic finite automaton or *DFA*, is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states*, Σ is the *alphabet*, $\delta : Q \times \Sigma \nrightarrow Q$ is the (partial) *transition function*, q_0 is the *initial state* and F is a subset of Q whose elements are *final states*. $|Q|$ is the number of states and $|\Sigma|$ is the number of elements in the alphabet, or *alphabet size*.

The *n-closure* of an alphabet is the set of all symbols that consist of concatenating n symbols from Σ. Σ^+ is the *plus-closure* of the alphabet, the set of symbols obtained by concatenating one or more symbols from Σ. We use the special alphabet $\mathbb{B} = \{0, 1\}$, the single bit alphabet. The n-closure of this alphabet allows us to define an alphabet of sequences of n bits: $\mathbb{B}^n$.

$|Q||\Sigma|$ is the theoretical *transition table size*. Note that since cells represent states, the minimum cell size is determined by the minimum space requirements to represent a state, which is in turn determined by the total number of states. Although stretching and jamming will change the number of states in an automaton we will assume that the transition table cell size does not change in either transformation. We expect that in most cases the practical effects of this assumption are unlikely to be significant. Preliminary benchmarking results indicate that our theoretical transition table size is indeed a good estimate for the real transition table size.

A transition in a DFA M from p to q with label a will be denoted by (p, a, q) where $(p, a, q) \in Q \times \Sigma \times Q$ and $q = \delta(p, a)$. We will also use the notation $((p, a), q) \in \delta$.

A *path* of length k in a DFA M is a sequence $\langle (r_0, a_0, r_1), \ldots, (r_{k-1}, a_{k-1}, r_k) \rangle$, where $(r_i, a_i, r_{i+1}) \in Q \times \Sigma \times Q$ and $r_{i+1} = \delta(r_i, a_i)$ for $0 \leq i < k$. The *string*, or *word* $a_0 a_1 \cdots a_{k-1} \in \Sigma^k$ is the *label* of the path.

The *extended transition function* of a DFA M, $\hat{\delta} : Q \times \Sigma^+ \nrightarrow Q$, is defined so that $\hat{\delta}(r_i, w) = r_j$ iff there is a path from r_i to r_j, labeled w.

A nondeterministic finite automaton, *NFA*, is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, defined in the same way as a DFA, with the following exception: $\delta : Q \times \Sigma \nrightarrow \mathcal{P}(Q)$ is the transition function. Note that $\mathcal{P}(Q)$ is the powerset of Q. For present purposes, ϵ-transitions can be ignored without loss of generality.

A transition in an NFA M from p to q with label a will also be denoted by (p, a, q) where $(p, a, q) \in Q \times \Sigma \times Q$ and $q \in \delta(p, a)$.

A path of length k in an NFA M is a sequence $\langle (r_0, a_0, r_1), \ldots, (r_{k-1}, a_{k-1}, r_k) \rangle$, where $(r_i, a_i, r_{i+1}) \in Q \times \Sigma \times Q$ and $r_{i+1} \in \delta(r_i, a_i)$ for $0 \leq i < k$.

In an NFA M, the extended transition function, $\hat{\delta} : Q \times \Sigma^+ \nrightarrow \mathcal{P}(Q)$, is also defined so that $r_j \in \hat{\delta}(r_i, w)$ iff there is a path from r_i to r_j, labeled w.

To stretch a transition we need to split up one symbol into 2 or more sub-symbols. Therefore, we conceive of alphabet elements as strings of *subelements* (typically bits). If alphabet element $a \in \Sigma$ has length $|a|$ then we number the subelements $a.0, \ldots, a.(|a| - 1)$. Thus, if $a = 0111$ then $a.0 = 0$, $a.1 = 1$, $a.2 = 1$ and $a.3 = 1$.

We use square brackets to denote a *substring* of an alphabet element. For $a \in \Sigma$, f a *factor* or *divisor* of $|a|$ (not to be confused with a factor or substring of a word) and $0 \leq i < f$, we use $a[f, i]$ to denote $a.(i\frac{|a|}{f}) \cdots a.((i + 1)\frac{|a|}{f} - 1))$. The length of the substring depends on the factor f used in stretching; every substring of symbol a has exactly length $|a|/f$. If there is no confusion, we sometimes leave out the factor f and use just $a[i]$ instead of $a[f, i]$. So for example, if FA_0 is stretched by a factor 2 and $a = 0111$ (so $|a| = 4$), then $a[0] = 01$ and $a[1] = 11$. If FA_0 is stretched by a factor 4 then $a[0] = 0$, $a[1] = 1$, $a[2] = 1$ and $a[3] = 1$.

By definition, a *word* is a string of symbols over an alphabet. We also number the individual symbols of a word. For the word $w \in \Sigma^k$, we number the individual symbols $w.0 \cdots w.(k - 1)$. Note that we use the same notation for the subelements of a single symbol as for those of a word. Which of these is meant will always be clear from the context.

3 Stretching and jamming

Stretching and jamming were first defined in [3] and [2]. There, successively more restrictive definitions of stretching and jamming were provided, starting from general definitions, via stretching or jamming by a fixed factor per automaton, to stretching and jamming at the bit level (i.e. relying on the fact that characters are typically encoded as or represented by bit strings). Here, we present the transformations succinctly yet informally, focusing on the last, most restricted and practical kind of stretching and jamming.

One way in which we can stretch an automaton is by splitting each transition into k sequential transitions. This stretching operation on a single transition is pictured in Figure 1.

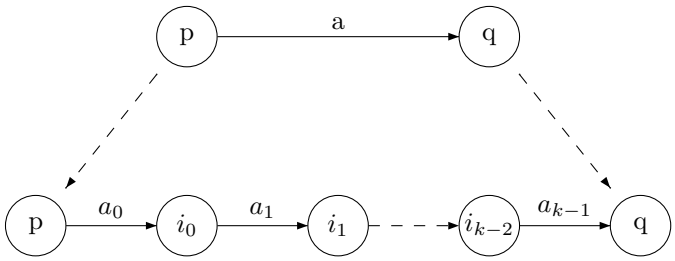


Figure 1. Stretching transition (p, a, q) into k sequential transitions.

In this example we see that transition (p, a, q) is stretched into k sequential transitions and $k - 1$ new states are introduced. In this sequence of transitions we call p and q the *original states*, and $i_0, \ldots, i_{k-2}$ the *additional intermediate states*.

Jamming is the inverse transformation, in which k sequential transitions are joined into a single transition. In Figure 1 this can be seen as performing a transformation in the opposite direction to stretching. This means that the intermediate states are removed. In the case of jamming we call these states *redundant intermediate states*.

If NFA FA_0 can be stretched into NFA FA_1, we call FA_1 *a stretch* of FA_0. The set of states of FA_1 consists of a subset S_1 of original states and a subset I of newly introduced additional intermediate states. Stretching requires

- An injection τ from the alphabet of FA_0, Σ_0, to Σ_1^+, the plus-closure of the alphabet of FA_1.
- A bijection φ between the original states of FA_0 and the original states of FA_1. This bijection connects the start states of FA_0 and FA_1 and defines a one-to-one relationship between the final states of both automata. It also ensures that for every transition from state p to q with label a in FA_0 there exists a path from $\varphi(p)$ to $\varphi(q)$ with label $\tau(a)$ in FA_1, which travels from $\varphi(p)$ in S_1 via a number of intermediate states to $\varphi(q)$ in S_1. The inverse of this property is also true.

We define jamming as the inverse transformation of stretching.

If NFA FA_0 is jammed into NFA FA_1 (FA_1 is *a jam* of FA_0) then FA_0 is a stretch of FA_1. The set of states of FA_0 consists of a subset S_0 of original states and a subset R of redundant intermediate states. These redundant intermediate states are removed by the jamming transformation.

The preceding requirements are very general, in that they allow every single transition to be stretched into a different number of sequential transitions—i.e. k can have a different value for every single transition of the original automaton. We therefore restrict stretching (and hence jamming) to stretching (jamming) by a fixed factor f:

If NFA FA_0 is stretched by a factor f into NFA FA_1, we call FA_1 *an f-stretch* of FA_0. This means that the relation τ specialises to a one-to-one relationship between the alphabet of FA_0 and the f-closure of the alphabet of FA_1. Furthermore, for each transition in FA_0 there are exactly f sequential transitions in FA_1.

Jamming by a factor f is defined analogously to stretching by a factor f. Note that, by definition, jamming by a factor f is not always possible. NFA FA_0 can only be jammed if there exists an NFA FA_1 which can be stretched into FA_0. In such a case, we call FA_0 an *f-jam* of FA_1. In Section 3.2 we consider this problem as well as some possible solutions to it.

As indicated in the introduction, alphabet symbols are typically represented as or encoded by a sequence of bits. We therefore only consider automata in which each element of the alphabet is a bit string. An *n-bit automaton* is an automaton whose alphabet consists of all the 2^n bit strings of length n.

Property 1. Let f be a factor of n. Then we can f-stretch the n-bit DFA FA_0 into NFA FA_1 in the following way:

- FA_1 is an $\frac{n}{f}$-bit NFA.
- There is a bijection between Σ_0 and Σ_1^f i.e. for every bit string of length n in Σ_0 there is a sequence of f bit strings of length $\frac{n}{f}$ in Σ_1^f and vice versa.
- For every transition in FA_0 there are f sequential transitions in FA_1, obeying the above bijection between Σ_0 and Σ_1^f for the labels of the transitions.

Of course, this specialisation of stretching is only allowed if n is divisible by f. In that case we call the DFA *f-stretchable*. Again, jamming is the inverse transformation: if an n-bit NFA is *f-jammable*, the resulting automaton is an *nf-bit* DFA.

Example 2. To illustrate the stretching of n-bit automata we give an example. The 2-bit DFA FA_0 of Figure 2 can be stretched by a factor 2 into the 1-bit NFA FA_1 of Figure 4. (We leave out the explicit definition of τ and ψ, both of which are implicitly defined by the figures.) Also, the 1-bit NFA FA_1 can be jammed into the 2-bit DFA FA_0.

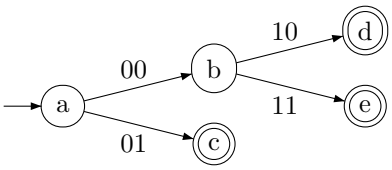


Figure 2. DFA FA_0

	00	01	10	11
a	b	c	-	-
b	-	-	d	e
c	-	-	-	-
d	-	-	-	-
e	-	-	-	-

Figure 3. Transition table of DFA FA_0

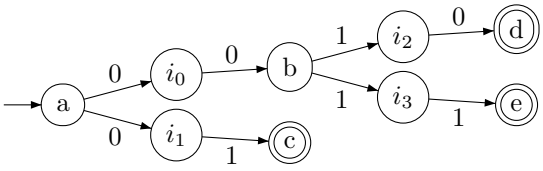


Figure 4. NFA FA_1, a 2-stretch of DFA FA_0

	0	1
a	$\{i_0,i_1\}$	-
i_0	{b}	-
i_1	-	{c}
b	-	$\{i_2,i_3\}$
i_2	{d}	-
i_3	-	{e}
c	-	-
d	-	-
e	-	-

Figure 5. Transition table of NFA FA_1

Note that in the previous example, we stretched a transition by using the most significant bit (MSB) first. For example, we stretched transition $(a, 01, c)$ into $(a, 0, i_1)$ and $(i_1, 1, c)$, taking the 0 first and then the 1. In practice, we will often use the least significant bit (LSB) first. This is due to the little-endianness of processor architectures such as the Intel x86 family. As this choice only has a minor influence on the stretching and jamming algorithms, it will not be considered further in this paper.

3.1 Stretching and nondeterminism

We can stretch NFAs as well as DFAs. In general, NFAs can be stretched and the result of such transformations will also be NFAs. Because DFAs are a subset of NFAs, the stretching of DFAs is automatically defined.

If we stretch a DFA, in some cases the resulting automaton may have more than one transition with the same label from a given state and therefore the result of stretching a DFA might be an NFA. This is apparent in the example, in which our choice of $\tau(00) = 0 \cdot 0^1$ and $\tau(01) = 0 \cdot 1$ causes two outgoing transitions with label 0 from state a to appear in FA_1 resulting from stretching FA_0. Therefore FA_1 is an NFA. Because of symmetry, if we jam certain NFAs the result will be a DFA.

We can easily prevent the introduction of such (additional) nondeterminism by checking for already created transitions during the stretching process, and following such transitions whenever possible. The stretching algorithm in Section 4 uses this approach.

3.2 Jamming and path lengths

Because of the symmetry in our definitions of stretching and jamming, jamming is only defined on the subset of NFAs that are stretched NFAs. This means that some NFAs are not jammable. For example, the NFA in Figure 6 clearly is not a stretched NFA and can therefore not be jammed according to the earlier definition.

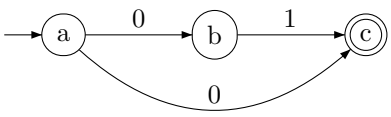


Figure 6. NFA $\mathrm{FA}_0 = (\{a, b, c\}, \mathbb{B}, \delta_0, a, \{c\})$

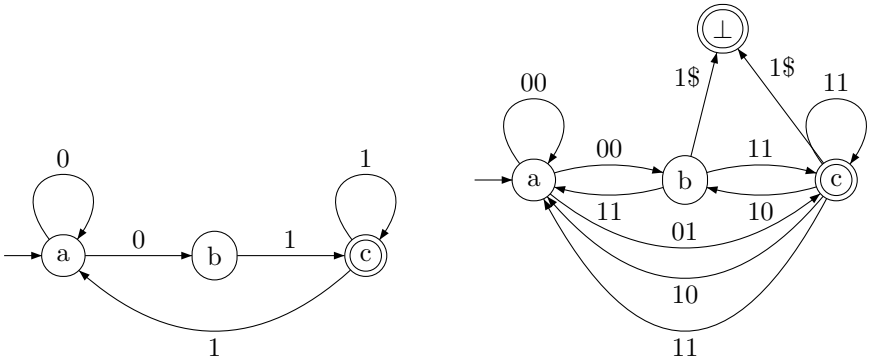
Let us now consider how to jam *any* NFA by a factor f. If we want to jam NFA FA_0 by a factor f into NFA FA_1, we essentially want to make f transitions in one step, instead of one transition at a time. We can achieve this if we find all paths of length f from the start state q_0 to states $t_0, \ldots, t_i$. Then we can add all transitions (q_0, w_j, t_j) to FA_1, where w_j is the label of the path from q_0 to t_j, $(0 \leq j \leq i)$. Next, we repeat the same process for states $t_0, \ldots, t_i$, and so on until no more new states are found.

Unfortunately, a transition added to the jammed NFA, FA_1, cannot always be guaranteed to represent a path whose length is f. For example, if there is an (extendable or non-extendable) path of length m $(m < f)$, in FA_0 from state s to a final state t with label w, the path with label w to this final state *must* be added in order to accept the word with label w in the jammed NFA. In the example in Figure 6, this occurs if we want to jam the automaton by say $f = 2$, as there is a path of length 1 from state a to final state c. We can solve this problem by creating a special final state $\perp$ in FA_1 with no outgoing transitions. Then we can add transition $(s, w\$^{f-m}, \perp)$ to FA_1, where $\$^{f-m}$ is used as *padding* to make the label $w\$^{f-m}$ exactly size f.

To illustrate this new approach to jamming an NFA, we give an example.

[1] Note that argument 00 represents a *single* symbol from the original automaton's alphabet.

Example 3. NFA FA_0 of Figure 7(a) can be jammed by a factor 2 into NFA FA_1 of Figure 7. We do this by finding all paths of length 2, as described before. For example, if we look at all such paths from state a in FA_0 we find a path with label 00 to state a itself, one with label 00 to state b, and one with label 01 to state c. If we look at the paths of length 2 from state b in FA_0 we find that on the path with label 11 ending in state c, c also occurs as a final state *inside the path*. Therefore, we have to add a transition to FA_1 from state b to state $\perp$ with label 1\$.



(a) NFA $FA_0 = (\{a, b, c\}, \mathbb{B}, \delta_0, a, \{c\})$ (b) NFA $FA_1 = (\{a, b, c, \perp\}, \mathbb{B}^2 \cup \{0\$, 1\$\}, \delta_1, a, \{c, \perp\})$

Figure 7. NFA FA_0 and NFA FA_1, a 2-jam of FA_0

The main disadvantage of the above solution is that substrings of input strings are not always recognised. For example, if the bit string 0111 is used as input for FA_0 of the previous example, both 0111 and its proper substrings 01 and 011 are accepted. However, if the same bit string is used as input for FA_1 only the bit string itself and the single proper substring 01 are accepted. The substring 011 is not accepted in this case because 2-bit value 01 leads to state c and subsequently 11 leads back to state a again—since we jammed by a factor of 2, no transition on the single original symbol 1 from state c exists and the match for 011 is not detected. This would make jamming unsuitable for cases where substrings must always be recognised e.g. for pattern matching. Various solutions to this secondary problem exist:

- One option is to not only process each jammed symbol $s = s_0 \dots s_{f-1}$, but also the $f - 1$ *padded prefix symbols* $s_0 \dots s_{f-1-m}\$^m$ for all m $(1 \leq m < f)$—i.e. the $f - 1$ padded prefixes of s's encoding. Clearly, this would completely negate any savings in string processing time one would hope to achieve in the first place by using jamming.
- A second option would be to mark states that have outgoing transitions on not only a jammed symbol s, but also on one or more of its $f-1$ padded prefix symbols. Upon reaching such a marked state, before continuing with regular processing of the next padded symbol from the input, say s, the state's outgoing transitions on padded prefix symbols should be processed to see whether they lead to accepting states.
- A third option is a variant of the second one, in which, instead of marking the state having the outgoing transition on jammed symbol s, the state to which this transition leads is marked. This state could then simply be annotated with exactly those of the padded prefix symbols derived from s that should be accepted.

Although our current implementation and benchmarking results do not take the above problems into account, it seems relatively straightforward to implement the solution with the third option indicated above. We plan to do so as part of more extensive benchmarking.

4 Algorithms

4.1 Stretch Algorithm

In this section we present an algorithm to stretch an n-bit DFA by a factor f. This algorithm can easily be generalised to an algorithm that can stretch NFAs.

As indicated before, to stretch an automaton, we stretch each single transition into f sequential transitions. Therefore, our algorithm must find each transition in the automaton. This is done using a variant of the well known *Breadth-First Search* (BFS) [1, Section 22.2]. Algorithm 5 starts in the start state and finds all outgoing transitions. All found transitions are stretched and added to the stretched DFA. This process is repeated for all states and transitions that are found by the algorithm. In the algorithm, set I represents the intermediate states added, queue Q is used to enqueue the 'grey' states as per the BFS algorithm, and set V represents the 'non-white' states as per that algorithm.

As indicated before, naive stretching of an automaton may introduce (additional) nondeterminism. In our algorithm, we prevent this from happening. If we have to add a transition with label a from a state p but such a transition exists already, then we do not add it but instead we take the existing transition. We continue with this until we have to add a transition that does not exist already. The innermost do-loop in Algorithm 5 takes care of this process.

Property 4. Let $\mathrm{FA}_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an n-bit DFA. Algorithm 5 will Stretch FA_0 by a factor f into $\frac{n}{f}$ *− bit* DFA $\mathrm{FA}_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1)$.

Algorithm 5 (**STRETCH**$(\mathrm{FA}_0, \mathrm{FA}_1, f)$)

Pre : $n \in \mathbb{Z}^+ \wedge f \in \mathbb{Z}^+ \backslash \{1\} \wedge n$ **mod** $f = 0$
Post : FA_1 is an f-stretch of FA_0
$[\![\ Q, V := \emptyset, \emptyset$
$; S_1 := S_0$
$; q_1 := q_0$
$; F_1 := F_0$
$; \Sigma_1 := \mathbb{B}^{\frac{n}{f}}$
$; \delta_1 := \emptyset$
$;$ *enqueue*(q_0, Q)
$;$**do** $Q \neq \emptyset \rightarrow$
$\quad p :=$ *dequeue*(Q)
$\quad ;$**for all** $q, a : q = \delta_0(p, a) \rightarrow$
$\quad\quad s_0^{pq} := p$
$\quad\quad ; i := 0$
$\quad\quad ;$**do** $\delta_1(s_i^{pq}, a[i]) \neq \emptyset \rightarrow s_{i+1}^{pq} := \delta_1(s_i^{pq}, a[i])$
$\quad\quad\quad\quad\quad\quad\quad\quad ; i := i + 1$

```
                od
               ;let s_{i+1}^{pq}, ..., s_{f-1}^{pq} be new states
               ;I := I ∪ {s_{i+1}^{pq}, ..., s_{f-1}^{pq}}
               ;s_f^{pq} := q
               ;for j := i to f − 1 → δ1 := δ1 ∪ {((s_j^{pq}, a[j]), s_{j+1}^{pq})} rof
               ;as q ∉ V → V := V ∪ {q}; enqueue(q, Q) sa
            rof
  od
]‖
```

Proof. We need to prove that algorithm 5 will correctly stretch FA_0.

- We have to prove there exists a bijection $\tau : \Sigma_0 \leftrightarrow \Sigma_1^f$. Because $\Sigma_1^f = (\mathbb{B}^{\frac{n}{f}})^f = \mathbb{B}^n = \Sigma_0$, there is a trivial bijection between the two sets. Intuitively, we can see that every bit string of length n can be constructed from a unique sequence of f bit strings of length $\frac{n}{f}$.
- After the algorithm is executed, $S_1 = S_0$, $q_1 = q_0$ and $F_1 = F_0$. Therefore there is an obvious bijection $\varphi : S_0 \leftrightarrow S_1$. Because the algorithm is based on the BFS algorithm, all states p that are reachable from the start state are found. For each such state p, all transitions (p, a, q) are found, for all symbols a and states q. According to our definition of stretching, if we find transition (p, a, q) we have to add one or more transitions such that $\hat{\delta}_1(p, w) = q$, with $\tau(a) = w$. We do this by ensuring that, after stretching the transition, a path from state p to state q with label $a[0]a[1]\cdots a[f-1]$ exists, where $a[0]a[1]\cdots a[f-1] = w$. In this path, state p and q are in set S_1 and the rest of the states are in the set of intermediate states I. DFA FA_1 might already have a path from state p with label $a[0], a[1]\cdots a[i]$, $(0 \leq i < f-1)$. If this is the case, this path is followed and only the remaining transitions are added to FA_1. □

During a stretch operation a number of intermediate states is inserted for every transition in the original automaton. Note that the number of states inserted by Algorithm 5 is not necessarily minimal, as exemplified in Figure 8.

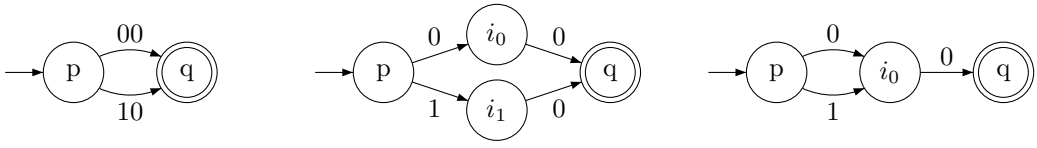


Figure 8. Minimisation after a stretching operation

This problem is a standard minimisation problem for DFAs. Therefore, after stretching a DFA, the resulting DFA can be minimised using one of the many DFA minimisation algorithms. It is also possible to minimise incrementally, after all outgoing transitions from a certain state have been stretched.

4.2 Jam Algorithm

If we want to jam NFA FA_0 by a factor f into NFA FA_1, we essentially want to make f transitions in one step, instead of one transition at a time. We can achieve this if we find all paths of length f from the start state q_0 to states $t_0, \ldots, t_i$. Then we can add all transitions (q_0, w_j, t_j) to FA_1, where w_j is the label of the path from q_0 to t_j, $(0 \leq j \leq i)$. Next, we repeat the same process for states $t_0, \ldots, t_i$, and so on until no more new states are found. To find all paths of length f from a give state p we can use a variant of the *Depth-First Search* (DFS) algorithm [1].

In the original DFS algorithm a set V is used for states that have already been found. If the search encounters a state that has already been found it does not explore the transitions out of that state. Because we need to find *all paths*, we also need to explore transitions out of states that have already been found. Therefore, instead of using a set V of states found, we use a variable c to indicate the current depth. If the search reaches depth f we stop searching.

Our algorithm uses a recursive procedure to search all paths of length f. The abstract algorithm can be described concisely and can be implemented easily. It is possible to do the search with a non-recursive procedure, using a stack to store all states found and the paths to these states.

Algorithm 6 (**JAM(**FA_0**,** FA_1**,** f**)**)

Pre : $n \in \mathbb{Z}^+ \wedge f \in \mathbb{Z}^+ \backslash \{1\}$
Post : FA_1 is an f-jam of FA_0
$[\![$ $Q := \emptyset$
$; S_1 := \{\perp\}$
$; q_1 := q_0$
$; F_1 := F_0 \cup \{\perp\}$
$; \Sigma_1 := \{x\$^{nf-ni} \colon x \in \mathbb{B}^{ni}, 1 \leq i \leq f\}$
$; \delta_1 := \emptyset$
$; enqueue(q_0, Q)$
$; S_1 := S_1 \cup \{q_0\}$
; **do** $Q \neq \emptyset \rightarrow q := dequeue(Q)$
$;$ JAM-PATH$(\mathrm{FA}_0, \mathrm{FA}_1, Q, q_0, q_0, f, 0, \epsilon)$
od
$]\!]$

Algorithm 7 (**JAM-PATH(**FA_0**,** FA_1**,** Q**,** r**,** p**,** d**,** c**,** l**)**)

Pre : $r, p \in S_0 \wedge d \in \mathbb{Z}^+ \wedge c \in \mathbb{N} \wedge c \leq d \wedge l \in \Sigma_0^c$
Post : All paths in FA_0 of length d, and starting in state r,
are jammed and added to FA_1
$[\![$ **if** $c < d \rightarrow$ **as** $p \in F_0 \rightarrow \delta_1 := \delta_1 \cup \{((r, l\$^{d-c}), \perp)\}$ **sa**
$;$**for all** $q, a : q \in \delta_0(p, a) \rightarrow$
JAM-PATH$(\mathrm{FA}_0, \mathrm{FA}_1, Q, r, q, d, c+1, la)$
rof

```
[[  c = d → as p ∉ S1 → S1 := S1 ∪ {p}; enqueue(p, Q) sa
          ;δ1 := δ1 ∪ {((r, l), p)}
  fi
]]
```

Property 8. Let $\mathrm{FA}_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an n-bit NFA. Algorithms 6 and 7 will jam FA_0 by a factor f into nf-bit NFA $\mathrm{FA}_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1)$.

Proof. The algorithm is based on the DFS algorithm, which is used to find all paths of length f from the start state q_0. All end states of the path, or in other words, all states that are at distance f from q_0, are added to the queue and for these states the same process is repeated recursively.

The label of each path is recorded in variable l, and every time a new transition (p, a, q) is found, the algorithm recursively calls itself with the new label la and new state q. This way, the algorithm recursively descends into the NFA.

Variable c indicates the current depth of the path, so if depth $d = f$ is reached, a path of length f from state r to state p with label l has been found. This means a new transition (r, l, p) can be added to the jammed NFA. If a final state is found on a path that does not have length $d = f$, a transition $(r, l, \perp)$ is added to FA_1, where r is the first state of the path, l the label, and $\perp$ is the special final state introduced.

Therefore, if a string is processed by the jammed NFA, it can travel any path to a final state as it would have done in the original NFA. This means the jammed NFA will accept the same set of strings as the original NFA.

We conclude this proof by proving the correctness of the construction of the alphabet. The original NFA FA_0 has an n-bit alphabet. The algorithm finds all paths of length f, but on each path a final state might be found. Therefore, paths of length $1, 2, \ldots, f$ can be found. Because each symbol in the original NFA is n bits long, a symbol in the new alphabet can have length n, $2n$,..., fn bits. The alphabet symbols of the jammed NFA must have bit-length nf, therefore padding with \$ symbols is added to the alphabet symbols if they do not have length nf. □

5 Implementation

The (improved) algorithms for stretching and jamming as presented in the preceding section were implemented in C++. This language was chosen for its flexibility and efficiency, in particular in combination with the Standard Template Library (STL).

Due to the similarities between DFAs and NFAs—both conceptual and hence in implementation as well—the implementation uses a single abstract base class FA, which in turn uses a class TransitionTable. Both are parameterised by the type of the cells of the transition table—i.e. State for DFAs and set<State> for NFAs. Parameterised class TransitionTable inherits from class Matrix, which in turn is further parameterised by the type of rows and columns to be used. Clearly, for a matrix used as a transition table, these correspond to states and alphabet symbols. In turn, such a matrix is naturally composed of vectors, hence the use of a parameterised type Vec.

Class FA, its derived classes and class TransitionTable have a method NextState which uses the current state (which TransitionTable keeps track of) and the next input symbol to advance to the next state(s). They also have methods to add transitions, resize the transition table, determine whether an automaton was created by stretching or jamming, and to return the size of the transition table or its density.

Classes DFA and NFA have a method to stretch by a factor of f, passed as a parameter together with a boolean indicating whether LSB and MSB should be reversed compared to the default (which on the Intel x86 architecture is LSB before MSB). Both methods return a new DFA/NFA object containing the stretched version of the original DFA/NFA. Class DFA also has a method to jam by a factor of f, with similar parameters. For NFAs, jamming has not yet been implemented.

Due to a technicality, jammed DFAs use additional data structures to represent transitions on any of the newly added padded symbols. Jammed DFAs are therefore represented by instances of class JammedDFA instead of those of class DFA. As jamming for NFAs is currently not implemented, no class JammedNFA is needed.

6 Benchmarking Results

We have performed preliminary benchmarking experiments using the implementations from the preceding section. Due to lack of space, we only report some of the results here. More details on the experiments, the environment and the results can be found in [2, Chapter 7]. The experiments were focused on stretching and jamming of DFAs. The experiments were used to compare DFAs' transition table memory use and their string processing time before and after stretching and jamming. To do so, random DFAs were generated using a (pseudo-)random number generator, and the computed memory use of their transition tables before and after stretching or jamming were compared. Random paths from these DFAs were generated to measure the DFAs' string processing time, both before and after stretching or jamming. The experiments were performed on an x86 family system running a version of Linux in single user mode, on which the implementations had been compiled using GCC 3.3.2. String processing time was measured using the CPU's time stamp counter. Memory use was computed based on the state set size, alphabet size, and resulting transition table cell size, as the STL vectors used for the transition table implementations do not guarantee the vector sizes not to be larger than needed to store just its current elements. (Note that memory use is still dependent on the number of intermediate states removed by jamming or inserted by stretching and therefore varies depending on the original DFA's structure.)

The benchmarking was performed with a range of parameters, and for each choice of parameter values used, 100 runs of the benchmark were performed and the mean and variance of memory use and processing time were collected. For both stretching and jamming, factors of 2, 4, and 8 were used, and alphabet sizes 2^1, 2^2, 2^4 and 2^8 were used. The generated DFAs had 10, 100 or 1000 states, with stretching (jamming) obviously increasing (decreasing) these numbers. Transition densities considered ranged from 1% - 100%. As input strings, strings of 8, 16, and 32 were considered. As we will see, not all combinations of these parameter values were used (some do not even make sense for all experiments).

Our theoretical analyses of the effects of the two transformations were reported in [3, Section 4] and [2, Chapter 3]. They show that stretching is expected to reduce memory use for low transition table densities: as few intermediate states will be introduced at such densities, this will be more than offset by the decrease in alphabet size, and hence the theoretical transition table size $|Q||\Sigma|$ will decrease. Clearly, jamming is primarily meant to reduce string processing time and expected to do so, while stretching is expected to increase it. In the remainder of this section we will see

how the benchmarking confirms these expectations and briefly discuss under which conditions stretching and jamming seem practically useful.

6.1 Stretching

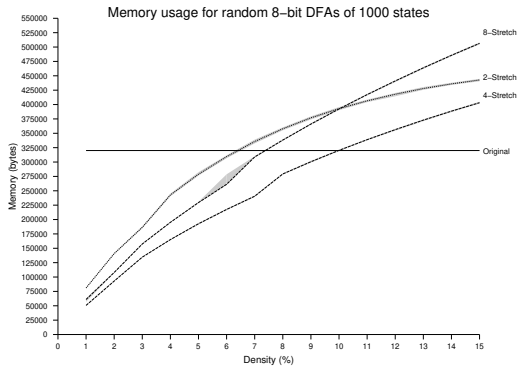


Figure 9. Memory usage results for stretching 8-bit DFAs

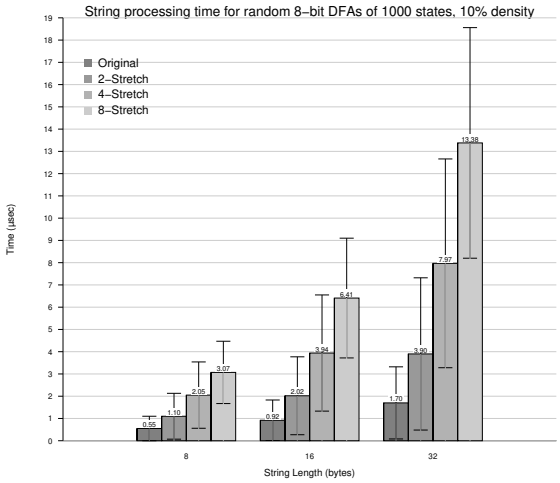


Figure 10. String processing time results for stretching 8-bit DFAs

Figures 9 and 10 show some of the results of the benchmarks for stretching. Figure 10 shows the influence of stretching on string processing time for a typical case, i.e. for 8-bit DFAs with 1000 states. Results for fewer states and higher densities are similar, and as expected density does not influence string processing time.

Figure 9 shows the impact on memory use, again for 8-bit DFAs with 1000 states. For 10 or 100 states, the memory use graphs are roughly the same, albeit with break even points in the $4-6\%$ and $5-9\%$ range, respectively. (Note the seemingly counter-intuitive high memory use of 8-stretches compared to 4-stretches. We suspect this is caused by memory allocation issues related to alignment and packing of objects, as well as the memory consumption reporting mechanisms used.) For 4-bit DFAs, the break even points are somewhat higher, ranging from $12-19\%$ depending on the number of states and stretch factor used. We therefore expect that benchmarking DFAs using larger symbols such as typically needed for applications of Unicode will

show that if such automata are of low density, memory use can be reduced manifold by stretching, even if just stretching by a factor of 2. Since applications of Unicode in automata tend to lead to large memory use because of the large number of symbols, this might be a worthwhile avenue to explore further.

6.2 Jamming

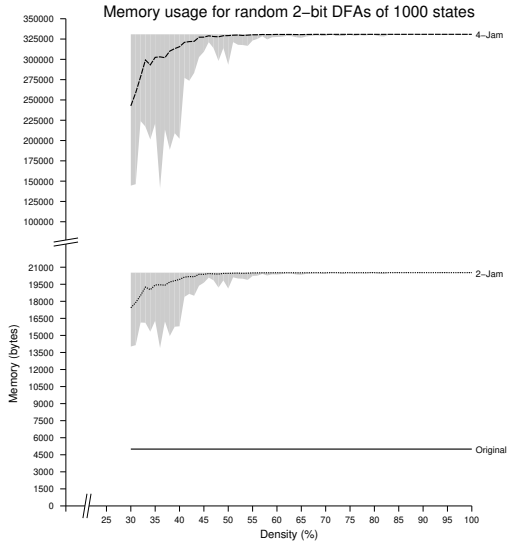


Figure 11. Memory usage results for jamming 2-bit DFAs

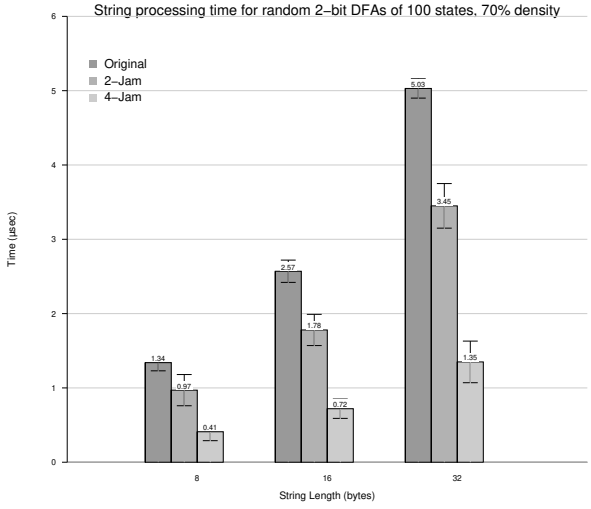


Figure 12. String processing time results for jamming 2-bit DFAs

Figures 11 and 12 show some of the results for jamming. Figure 11 shows the impact of jamming on memory use, for the case of 2-bit DFAs of 1000 states. Results for fewer states and smaller alphabets are again roughly similar. What stands out immediately is the horizontal upper bound in the range of memory usage for all three

factors of jamming. This is easily explained though, as even though jamming has the potential to reduce the number of states by removing redundant intermediate states, in the worst case no such states exist, while the alphabet size is always increased by jamming. The *average* memory use approaches the upper bound as density increases, as the likelihood of states being redundant decreases with an increase in transition density. (Note that the fact that transition density does not start at 0% but starts above 25% is due to our requirement that the DFAs generated be connected.)

Figure 12 clearly shows how jamming improves string processing time by $40-45\%$ when 2-jamming and reduces it by a further $2.3-2.6$ times when comparing this to 4-jamming—i.e. by $3.2-3.7$ times in total when going from the original 2-bit symbols to 8-bit symbols. As the DNA alphabet corresponds to a 2-bit alphabet, the results show the potential of using jamming for such an alphabet, provided memory use is not an issue at all or transition density is not too high.

7 Conclusions and Future Work

We have presented stretching and jamming and given improved versions of the algorithms from [3], which prevent nondeterminism from being introduced during stretching. Furthermore, we have sketched solutions to prevent jammed automata from no longer detecting all matches the underlying original automata would detect. The preliminary benchmarking studies reported here and in [2], for the case of DFAs, confirm our theoretical analysis from [3]: while jamming increases memory use, particularly for DFAs with a high transition table density, it reduces string processing time drastically. Conversely, stretching increases string processing time considerably, but reduces memory use for DFAs with a low transition table density.

Our benchmarking results can be extended in multiple directions. Although the benchmarking we performed already covers DNA and protein alphabets (which need between 2 and 5 bits to represent a symbol), it does not cover Unicode alphabets (needing up to 32 bits to represent a symbol). Extending the experiments to such alphabets could therefore be considered. In particular, with an eye on when and how to apply the transformations in practice, experiments with DFAs from practical applications in DNA and Unicode text processing should be performed. In particular, it should be investigated whether the increased memory use when jamming DNA automata is acceptable when dealing with the amount of data resulting from current high throughput DNA sequencing technologies. Furthermore, the experiments could be extended to cover NFAs in addition to DFAs, using the solutions we sketched for the match detection problem.

There are a number of additional problems that can be investigated further. We only considered stretching or jamming the complete transition table. Transforming only a small part of the transition table, in other words local stretching and jamming, is an interesting problem for further research. So is the use of stretching and jamming when sparse matrices are used to represent low density transition tables.

References

1. T. Cormen, C. Leiserson, and R. Rivest: *Introduction to algorithms*, McGraw-Hill, 2001.
2. A. de Beijer: *Stretching and jamming of automata*, Master's thesis, Eindhoven University of Technology, 2004.

3. N. De Beijer, B. W. Watson, and D. G. Kourie: *Stretching and jamming of automata*, in SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology, South African Institute for Computer Scientists and Information Technologists, 2003, pp. 198–207.
4. J. Hopcroft, R. Motwani, and J. Ullman: *Introduction to automata theory, languages, and computation*, Addison-Wesley, 2001.
5. B. Melichar and J. Skryja: *On the size of deterministic finite automata.*, in Proc. of the Sixth International Conference on Implementation and Application of Automata (CIAA 2001), B. Watson and D. Wood, eds., vol. 2494 of LNCS, Pretoria, South Africa, July 2001, pp. 202–213.