Weak factor automata: The failure of failure factor oracles?

Loek Cleophas[*], Derrick G. Kourie[†], Bruce W. Watson[†]

[*]FASTAR Research Group (http://www.fastar.org), Department of Computer Science, University of Pretoria, Private Bag X20, 0028 Hatfield, Pretoria, Republic of South Africa, `http://www.cs.up.ac.za`
[†]FASTAR Research Group (http://www.fastar.org), Department of Information Science, Stellenbosch University, Private Bag X1, 7602 Matieland, Republic of South Africa, `http://www.informatics.sun.ac.za`

ABSTRACT

In indexing of, and pattern matching on, DNA and text sequences, it is often important to represent all *factors* of a sequence. One efficient, compact representation is the *factor oracle* (FO). At the same time, any classical deterministic finite automaton (DFA) can be transformed to a so-called failure one (FDFA), which may use failure transitions to replace multiple symbol transitions, potentially yielding a more compact representation. We combine the two ideas and *directly* construct a *failure factor oracle* (FFO) from a given sequence, in contrast to ex post facto transformation to an FDFA. The algorithm is suitable for both short and long sequences. We empirically compared the resulting FFOs and FOs on number of transitions for many DNA sequences of lengths $4-512$, showing gains of up to 10% in *total number* of transitions, with failure transitions also taking up less space than symbol transitions. The resulting FFOs can be used for indexing, as well as in a variant of the FO-using backward oracle matching algorithm. We discuss and classify this pattern matching algorithm in terms of the keyword pattern matching taxonomies of Watson, Cleophas and Zwaan. We also empirically compared the use of FOs and FFOs in such backward reading pattern matching algorithms, using both DNA and natural language (English) data sets. The results indicate that the decrease in pattern matching performance of an algorithm using an FFO instead of an FO may outweigh the gain in representation space by using an FFO instead of an FO.

KEYWORDS: algorithmics, dictionary, pattern matching, DNA sequences

CATEGORIES: H.3.1, F.2.2, J.3

1 INTRODUCTION AND BACKGROUND

In recent decades, growing quantities of DNA and protein sequence data have been and are being generated, increasing the need for compact and efficient representations to store and search such data [1, 2]. Similar developments can be found for natural language processing as well.

One important data representation is that of all *factors* of a given *sequence* or *string* of characters (nucleotides or proteins). Factors are *contiguous* subsequences of a given sequence or string, and they play a role in at least two important algorithmic problems related to string processing in general and DNA processing in particular. Firstly, factors that occur multiple times within a given DNA sequence may indicate interesting genetic phenomena such as tandem repeats and micro-satellites [3]. To detect such factors, an *index* representing all the factors of a given sequence can be constructed in such a way that it provides information about repeated factors, including where they are located in the given sequence [1, 4].

Secondly, one is often interested in finding the occurrences of a given relatively short *pattern* or *keyword* in a long *subject* string on the fly, because the subject string may not be available at pre-processing time, or because one wants to find the occurrences of the pattern in multiple such subjects.[1] Among the most efficient approaches for such *pattern matching* are those that require to know that a subsequence read in the subject text is *not* a factor of the pattern [5, 6, 7, 8]. For a given position in the subject, such efficient pattern matching algorithms read backward from that position, trying to match (the reverse of) the pattern against what is read; if it can be determined that what has been read backward so far is not a (reverse) factor of the pattern, certainly no match of the pattern will be found by reading further backward, and a forward *jump* or *shift* to another position in the subject can be made, using the knowledge of what was read to allow shifts that skip over some positions of the subject string.

The *factor oracle* (FO) (and its variations) provides one particular data structure for efficient factor representation [9]. In fact, the language recognized by an FO constructed for a sequence or string p may recognize slightly more than just the factors of p, but that does not pose a problem per se for the two algorithmic problems mentioned:

Email: Loek Cleophas `loek@fastar.org`, Derrick G. Kourie `derrick@fastar.org`, Bruce W. Watson `bruce@fastar.org`

[1]For simplicity's sake, we consider the case of a single subject and a single pattern here.

- For pattern matching, reading backward slightly further than necessary does not influence correctness, and the idea is that the loss in efficiency by doing so is traded off for the gain in memory space by using an FO instead of an exact factor automaton.
- For indexing, various tricks can be applied during oracle construction to distinguish non-factors from factors [1].

The benefits of using oracles instead of exact factor automata lay in a number of properties of the factor oracle: it has just $m + 1$ states, has between m and $2m - 1$ transitions, is acyclic, and for each state all transitions leading to the state are on the same symbol (i.e. the automaton is *homogeneous*). These properties allow for a more compact, efficient representation than an exact factor automaton, which in most cases will have more states and transitions (while also being acyclic).

In the context of string pattern matching automata, so-called *failure transitions* arose in the 1970s [10, 11], with [12] considering further optimizations, and finally [13] giving a general definition of a *Failure Deterministic Finite Automaton* (FDFA) and a general construction algorithm for obtaining one given a complete DFA. Failure transitions differ from symbol transitions and resemble epsilon transitions in not consuming an input symbol, but may only be taken if the current state has no symbol transition for the given input symbol. If two states in a classical finite automaton share a set of outgoing symbol transitions, under certain conditions one such transition set may be replaced by a failure transition from the one to the other state. This leads to space savings at the cost of extra string processing time at runtime, since zero or more failure transitions followed by a symbol transition are taken, versus a single symbol transition in the classical DFA case. The FDFA construction algorithm given in [13] takes a complete DFA as input, and uses a heuristic involving the construction of a formal concept lattice to replace sets of symbol transitions by failure arcs. Björklund et al. [14] recently showed that even without changing the state set from DFA to FDFA, the problem of minimizing the number of transitions by replacing symbol transitions by failure transitions is NP-complete, but can be approximated efficiently within a factor of $\frac{2}{3}$.

Due to the above results, a different approach from *ex post facto* replacement of sets of symbol transitions by failure transitions was taken by the authors of [15]. They adapted the FO construction algorithm of [16] to *directly* create an FDFA called the *failure factor oracle* (FFO). Initially this approach, as well as a similar approach for the factor storacle of [17], was applied to compare the sizes of factor oracles, factor storacles, and their respective directly constructed failure alternatives [15]. This was done on generated and English language patterns (dictionary words). We recently discovered that for the FFO, the construction algorithm used in the original manuscript of [15] was suitable for short keywords only, possibly leading to cycles and live-lock situations even during construction of the automata, but only for longer keywords than those used in that manuscript.

We therefore adapted the original construction algorithm, using an *improved* construction algorithm for directly constructing an FFO, an algorithm which is suitable for sequences of any length, including those typically seen in the context of DNA processing. Both of these direct failure oracle construction algorithm variants do not necessarily result in FDFAs that are *language equivalent* to the FO for the same keyword, but the improved algorithm *does* terminate regardless of sequence length, and results in automata recognizing at least all factors of the given sequence. We in fact show that the resulting FFOs satisfy most of the important properties of the FO: recognizing at least all factors of the given sequence, having $m + 1$ states, having between m and $2m - 1$ transitions, and being homogeneous (apart from failure transitions). We also show however that both our original construction and our improved construction do not always result in an acyclic automaton, but that the introduction of a cycle is rare and in case of the improved construction algorithm does not prevent the use of the resulting FFO for pattern matching.

We empirically compared the resulting automata on number of transitions, using DNA sequences with lengths in the range $4 - 512$ as input to the FO construction algorithm as well as to the improved FFO construction algorithm. The results show gains of up to 10% in *total number* of transitions, with failure transitions also needing less space to present than symbol transitions, thus leading to larger savings in representation size. Similar results for natural languages (English) were initially reported in [15] and are discussed and compared as well. Preliminary results on sequence processing runtimes when using FFOs originally showed these to be multiples of those when using FOs, i.e. the gain in space is traded off for a substantial loss in running time, but partial memoization already leads to drastic runtime improvements. With such partial memoization, the FFOs were also used in a variant of the FO-using backward oracle matching algorithm [9]. We discuss and classify this pattern matching algorithm in terms of the keyword pattern matching taxonomies of Watson, Cleophas, and Zwaan [7, 18, 19]. We also empirically compared the use of FOs and FFOs in such backward reading pattern matching algorithms, using both DNA and natural language (English) data sets. The results indicate that the decrease in pattern matching performance of an algorithm using an FFO instead of an FO unfortunately may outweigh the gain in representation space by using an FFO over an FO. The resulting research was previously presented as a conference paper as [20]. In the current article, we extend the results with empirical evaluation of FFOs versus FOs in practical string/sequence processing settings, i.e. when used in the efficient pattern matching algorithms mentioned above (so-called *backward (reading) pattern matching algorithms*).

The rest of this paper is organized as follows. The next subsection gives some basic definitions. Section 2 briefly recalls the suffix-based factor oracle construction algorithm of [16] as well as some of the properties of the resulting FOs. Section 3 briefly recalls FDFAs and related definitions, before Section 4 presents our improved suffix-based FFO construction algorithm, generalizing the algorithm from an earlier version of [15], and showing which properties of the original FO hold and which do not hold for this FFO. Section 5 presents empirical results contrasting the size of the FO and FFO for DNA sequences of various lengths. Section 6 discusses backward oracle pattern matching, its adaptation to FFOs instead of FOs, as well as the algorithm's classification in a taxonomy of keyword pattern matching algorithms. Section 7 discusses the results of benchmarking the effect of using FFOs versus FOs in both rudimentary sequence processing and efficient pattern matching algorithms. Finally, in Section 8 we provide some concluding remarks.

1.1 Preliminaries

A *string* $p = p_1...p_m$ of length m is a sequence of characters from an alphabet V. A string u is a *factor* (resp. *prefix*, *suffix*) of a string v if $v = sut$ (resp. $v = ut$, $v = su$), for $s, t \in V^*$. We will use $\mathbf{pref}(p)$, $\mathbf{suff}(p)$ and $\mathbf{fact}(p)$ for the set of prefixes, suffixes and factors of p respectively. A prefix (resp. suffix or factor) is a *proper* prefix (resp. suffix or factor) of a string p if it does not equal p. These notions are extended to a set of strings $P = \{p_1, p_2, ..., p_r\}$ in the usual way. We will use $\leq_p$ to denote that a string is a prefix of another string. For $p \neq \varepsilon$ we use $p\upharpoonright n$ and $p\upharpoonleft n$ for p's rightmost and leftmost $min(n, |p|)$ characters respectively. Similarly, we use $p\downharpoonright n$ and $p\downharpoonleft n$ for p minus its rightmost or leftmost $min(n, |p|)$ characters respectively.

2 SUFFIX-BASED CONSTRUCTION OF THE FACTOR ORACLE

The factor oracle construction algorithm presented in [16], repeated as Algorithm 1 here, essentially consists of two parts: first, a 'skeleton' automaton having exactly $m+1$ states and m transitions and recognizing $\mathbf{pref}(p)$ is constructed; secondly, a path is constructed for each of the suffixes of p in order of decreasing length, such that eventually at least $\mathbf{pref}(\mathbf{suff}(p)) = \mathbf{fact}(p)$ is recognized. If such a suffix of p is already recognized, no transition needs to be constructed. If on the other hand the complete suffix is not yet recognized there is a longest recognized prefix of such a suffix.

A transition on the next, non-recognized symbol is then created, from the state in which this longest recognized prefix of the suffix is recognized, to the unique state from which recognition of the remainder of that suffix is certain to lead to state $m + 1$.

Figure 1 shows the FO for *abbc*, while Figure 2 shows the one for *abcaabaababc*.

Factor oracles are known to have a number of important properties [9, 16]:

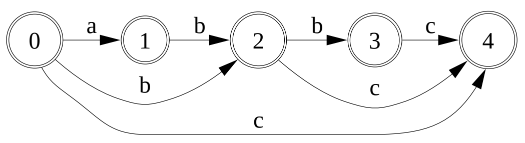

Figure 1: Factor oracle (with initial state 0) for *abbc*, with 7 transitions, recognizing $abc \notin \mathbf{fact}(p)$.

- They recognize at least all factors of the keyword.
- They are acylic. This property ensures that the only keyword of length m recognized is p itself, simplifying the use of FOs in backward pattern matching as mentioned in the introduction: if the algorithm has read m characters backwards successfully using the FO, we are certain that a match of p (in reverse) has been detected.
- They have exactly $m+1$ states (vs. typically more than $m + 1$ for exact factor automata).
- They have between m and $2m - 1$ transitions (vs. typically more for exact factor automata).
- They are homogeneous, meaning that for each state, all transitions to the state are on one and the same symbol.

These properties ensure that FOs can be represented efficiently compared to exact factor automata and DFAs in general. It should be noted that the above algorithm is $\mathcal{O}(m^2)$ and that an $\mathcal{O}(m)$ construction algorithm for FOs exists, given in [9]. The reasons we focus on the first algorithm here are threefold: this algorithm is much easier to understand, makes most of the mentioned properties obvious, and it formed the basis for our improved FFO construction in Section 4. It is an open question whether the $\mathcal{O}(m)$ algorithm of [9] could be adapted to yield the same FFOs as constructed by our construction.

3 FAILURE DETERMINISTIC FINITE AUTOMATA

As discussed in the introduction, an FDFA is a DFA, but may have so-called failure transitions in addition to normal symbol transitions. Such a transition is allowed to be taken if and only if processing of the current symbol from the current state using a symbol transition is not possible, and may replace multiple symbol transitions of a classical DFA, saving on the number of total transitions as well as transition representation space. Formally, $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ is an FDFA if $\mathfrak{f} : Q \to Q$ is a possibly partial function and $\mathcal{D} = (Q, \Sigma, \delta, F, s)$ is a DFA [13] (with state set Q, alphabet Σ, transition function $\delta \subseteq Q \times \Sigma \to Q$, final state set $F \subseteq Q$ and start state $s \in Q$).

As in [13], $\Sigma_q = \{a : \delta(q, a) \neq \bot\}$ is used to denote symbols labelling out-transitions of state q, and $\not\Sigma_q$ for $\Sigma \setminus \Sigma_q$ i.e. the symbols *not* labelling any out-transition of state q.

The *right language of an FDFA's state*, say of state q in FDFA $\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$, denoted by $\overrightarrow{\mathcal{L}}(\mathcal{F}, q)$,

Algorithm 1 Build_Oracle($p = p_1p_2...p_m$)

```
1: for $i$ from 0 to $m$ do
2:     Create a new final state $i$
3: for $i$ from 0 to $m-1$ do
4:     Create a new transition from $i$ to $i+1$ on symbol $p_{i+1}$
5: for $i$ from 2 to $m$ do
6:     Let the longest path from state 0 that spells a prefix of $p_i...p_m$ end in state $j$ and spell out $p_i...p_k$
       $(i-1 \le k \le m)$
7:     if $k \ne m$ then
8:         Build a new transition from $j$ to $k+1$ on symbol $p_{k+1}$
```

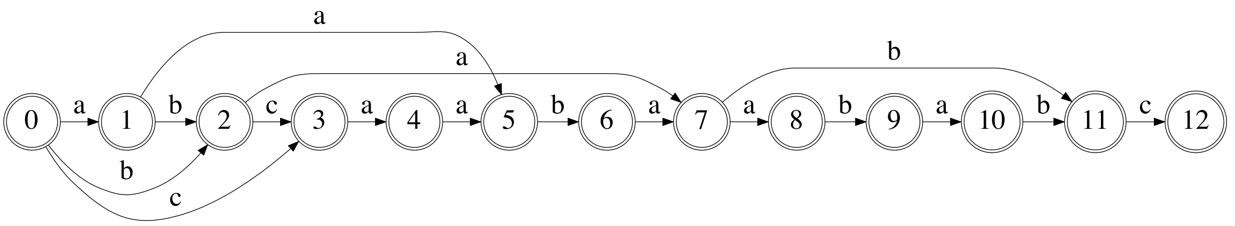

Figure 2: Factor oracle (with initial state 0) for *abcaabaababc*, with 17 transitions, and recognizing for example *bcaababc* $\notin$ **fact**(p).

is defined in [13] as $\overrightarrow{\mathcal{L}}(\mathcal{F}, q) = \overrightarrow{\mathcal{L}}_\delta(\mathcal{F}, q) \cup \overrightarrow{\mathcal{L}}_\mathfrak{f}(\mathcal{F}, q)$, where

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

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

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

Given the definition of right languages of FDFA states, the *language of an FDFA* $\mathcal{F}$, denoted by $\mathcal{L}(\mathcal{F})$, is then defined in [13] as $\overrightarrow{\mathcal{L}}(\mathcal{F}, s)$, where s denotes the start state of $\mathcal{F}$.

The definition of FDFAs leads to potential complications in the presence of cycles of failure transitions. More precisely, such failure cycles are problematic if, for one or more symbols, no state on the failure cycle has an out-transition labelled by this symbol. In [13] such a cycle is called a *divergent failure cycle*, and the FDFA construction algorithm presented there ensures these are not created.

4 SUFFIX-BASED CONSTRUCTION OF A FAILURE FACTOR ORACLE

In [15], failure transitions were introduced *during* construction of FFOs. The original version of the construction algorithm discussed there does not result in any cycles for short keywords such as those considered in the experiments discussed in [15], but the conjecture that the resulting automata are always acyclic, and as a result that no (divergent) failure cycles are introduced, turned out not to be true in general: while experimenting with longer keywords, we encountered situations where backward failure transitions are created. While this is not problematic per se, it also turned out that under certain (relatively rare) conditions, a pair of states might end up with failure transitions between them, and with the construction algorithm getting into a live-lock because neither state has an outgoing symbol transition on the next symbol to be processed. To solve this problem, we adapted the original algorithm to the improved FFO construction algorithm, also presented below. Several solutions to the problem were considered:

- Detecting such a live-lock and explicitly failing to construct an FFO in such cases. This has the major disadvantage that for some keywords, the algorithm cannot construct an automaton (although a classical FO could be used in such cases).
- Detecting the creation of a failure cycle, created by inserting a failure transition to say a state k, and in such cases constructing an appropriate *symbol* transition to state $k+1$ instead. This seems computationally expensive because of the need to detect cycles.
- Preventing the creation of failure cycles without the need for cycle detection, by only creating *forward* failure transitions; in other cases, instead of

Algorithm 2 Build_Failure_Oracle($p = p_1p_2...p_m$)

```
1:  for $i$ from 0 to $m$ do
2:      Create a new final state $i$
3:  for $i$ from 0 to $m-1$ do
4:      Create a new transition from $i$ to $i+1$ on symbol $p_{i+1}$
5:  for $i$ from 2 to $m$ do
6:      Let the longest recognized prefix of $p_i...p_m$ be recognized in state $j$ and spell out $p_i...p_k$ $(i-1 \leq k \leq m)$,
        and let the longest failure transition path from $j$ end in state $j'$
7:      if $k \neq m$ then
8:          if $k > j'$ then
9:              Build a new failure transition from $j'$ to $k$
10:         else
11:             Build a new symbol transition on symbol $p_{k+1}$ from $j'$ to $k+1$
```

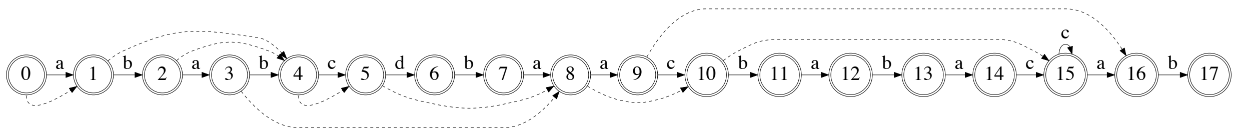

Figure 3: Failure factor oracle (with initial state 0) for *ababcdbaacbabacab*, having a self-transition on symbol c for state 15.

creating a *backward* failure transition to a state k, create an appropriate *symbol* transition to state $k+1$ instead. This clearly prevents failure cycles and hence divergent failure cycles from being created, without being computationally expensive. This option is therefore the one chosen in Algorithm 2.

For the inner **if**...**else** in Algorithm 2, note that $k \neq j'$ since otherwise $p_i...p_k$ is not the longest recognized prefix of $p_i...p_m$. Hence, for the **else**-case $k < j'$ and $k+1 \leq j'$ so the constructed symbol transition is either a backward one or a self-loop.

Figure 4 shows the FFO for *abbc*, while Figure 3 shows the one for *ababcdbaacbabacab*.

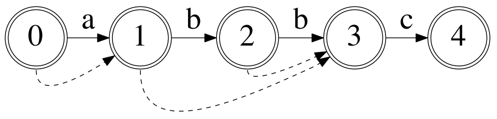

Figure 4: Failure factor oracle (with initial state 0) for *abbc*, with 7 transitions, recognizing $abc \notin \mathbf{fact}(p)$.

The FFOs constructed by Algorithm 2 have a number of important properties:

- Like FOs, they recognize at least all factors of the input keyword. This is easy to see: lines $1-4$ of the algorithm construct a 'skeleton' recognizing $\mathbf{pref}(p)$, while the **for**-loop of lines $5-11$ considers every non-empty proper suffix of p and ensures it is recognized as well—and since all states are final, all prefixes of such a suffix are recognized as well.
- Like FOs, they have exactly $m+1$ states (created in lines $1-2$).
- Like FOs, they have between m and $2m-1$ transitions: m are created in lines $3-4$, while at most one is created per non-empty proper suffix in the **for**-loop of lines $5-11$, and there are $m-1$ such suffixes.
- They are *weakly homogeneous*, i.e. for each state, all *symbol* transitions to the state are on one and the same symbol, but failure transitions to the state may exist as well (unlike for FOs).
- Unlike FOs, they are not in general acyclic. Intuitively, this is relatively easy to see: it may be that due to existing failure transitions, j' may be larger than k. Figure 3 shows an example of an FFO with a cycle.
- Due to the possibility that j' may be larger than k, it may also occur that a transition needs to be constructed while this is not the case for the corresponding FO. As a result it is also possible, somewhat counterintuitively, and unlike for FDFAs constructed from DFAs ex post facto, that an FFO has slightly more transitions than the corresponding FO. Figures 2 and 5 show the case of keyword *abcaabaababc*, for which the FFO has one more transition than the FO.

As we will see in the next section, both of the last two situations—i.e. FFOs with cycles, and FFOs that are larger than the corresponding FOs—are fairly rare in practice, and even in the rare case that the FFO is larger, it is larger by a small number of transitions—typically just one.

It should be noted that, as mentioned in the introduction, for any keyword, the FFO constructed by our direct construction on the one hand and the FO on the other hand may accept (slightly) different languages; for example, the FFO in Figure 3 accepts *all* strings of the form c^n for $n >= 2$, while the corresponding FO, not depicted, has no cycles and therefore cannot accept all such strings.

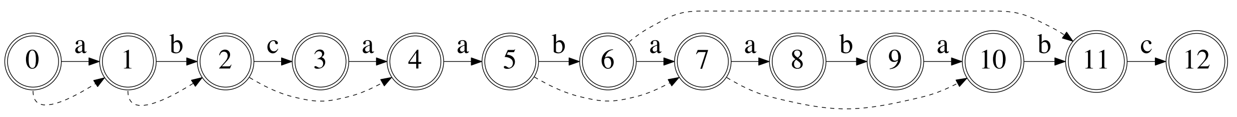

Figure 5: Failure factor oracle (with initial state 0) for *abcaabaababc*, having 18 transitions.

Due to the potential presence of cycles in FFOs, another property of the FO is potentially violated, namely that the automaton accepts exactly one string of length $m = |p|$, namely the keyword itself. This property is relevant in the efficient, backward-reading keyword pattern matching algorithms mentioned in the introduction: assuming the property holds, if m characters have successfully been read backward from a given position, one is certain to have detected a match of p. This is no longer guaranteed in the presence of cycles. In case any non-forward symbol transition was added during FFO construction for a particular keyword, the pattern matching algorithm using this automaton will therefore need to keep track of whether any non-skeleton transition has been used; if none have been used, and m characters have successfully been read backward, a match of p is still guaranteed to have been detected. Such use of any non-skeleton transition is easily detected.

5 EMPIRICAL RESULTS ON SIZE OF (FAILURE) FACTOR ORACLES

We implemented the FO and FFO construction algorithms in Java, and ran experiments using these constructions to obtain information about the distribution of the sizes of the resulting automata, and to use the resulting automata for preliminary benchmarking the running times when using these automata for (rudimentary) string processing. (As construction is typically done only once, and hence much more rarely than string processing using the constructed automata, construction times were not measured.) The benchmarks were run on an 1.7 Ghz Intel Core i5 with 4 GB of 1333 Mhz DDR3 RAM, running OS X 10.8.4.

The data set used was that of the DNA sequences of the various chromosomes of *Saccharomyces cerevisiae (strain S288c)* as available at [21], concatenated across chromosomes to yield a single sequence of length 12156677.[2] For each of $m = 4, 8, ..., 2048$, the resulting sequence was cut into subsequences of length m, and duplicates were removed from the resulting datasets. For small values of m (i.e. $m = 4, 8$), (almost) all possible subsequences on the alphabet $\Sigma = \{A, C, G, T\}$ occur, while for larger values of m far fewer of the $|\Sigma|^m$ possible sequences occur in the data set.

Figure 6 depicts the distribution and average for the *number of transitions* of the FO and the FFO, for $m = 4, 8, ..., 512$ respectively. (Results for $m = 1024, 2048$ are similar and omitted.) The results clearly show different distributions for the two automata kinds, with the average (depicted using dotted line) of the various distribution curves for the FFOs being smaller than for the FOs of the same keyword length. The distribution of the *difference* in number of transitions of the FFO and FO for the respective keywords in the data set are depicted in Figure 8, using a logarithmic scale on the y-axis, and in percentages of the total number of automata for keywords of a given length. The *average* savings in number of transitions between FOs and FFOs are shown as a percentage of the average number of FO transitions in Figure 7, ranging from a mere 1.5% for $m = 4$ to around 10% for most other cases. In addition to these savings in number of transitions, many transitions for the FFO cases will be failure transitions, which are cheaper to represent than symbol transitions, yielding additional savings above and beyond the percentages indicated.

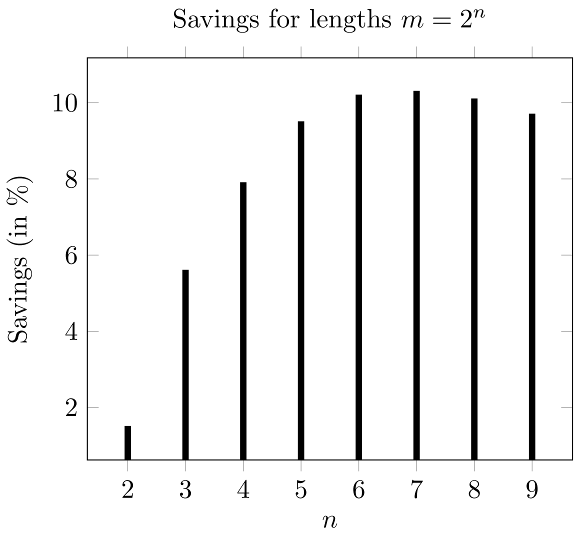

Figure 7: Average savings in number of transitions for failure factor oracles versus factor oracles for keyword lengths $m = 2^n$ for $n = 2, ..., 9$.

We also measured the occurrence among the test set of cases with one or more backward or self-symbol transitions created. These turn out to become less rare with increasing keyword length. For $m = 4$, 0 cases occur; for $m = 8$, 0; for $m = 16$, 1 (making $< 0.001\%$ of the unique sequences of that length in the data set); for $m = 32$, 47 (ca. 0.01%); for $m = 64$, 119 (0.06%); for $m = 128$, 161 (0.17%); for $m = 256$, 144 (0.30%); for $m = 512$, 106 (0.45%); for $m = 1024$, 79 (0.67%); for $m = 2048$, 55 (0.93%). However, the results comparing the size difference between the FO and the FFO for each keyword also indicate that it is very rare for an FFO to be larger than the FO for the same keyword—see the rare -1 size differences in

[2]Although such concatenation may introduce artefacts, this is not a problem in our *benchmarking* context.

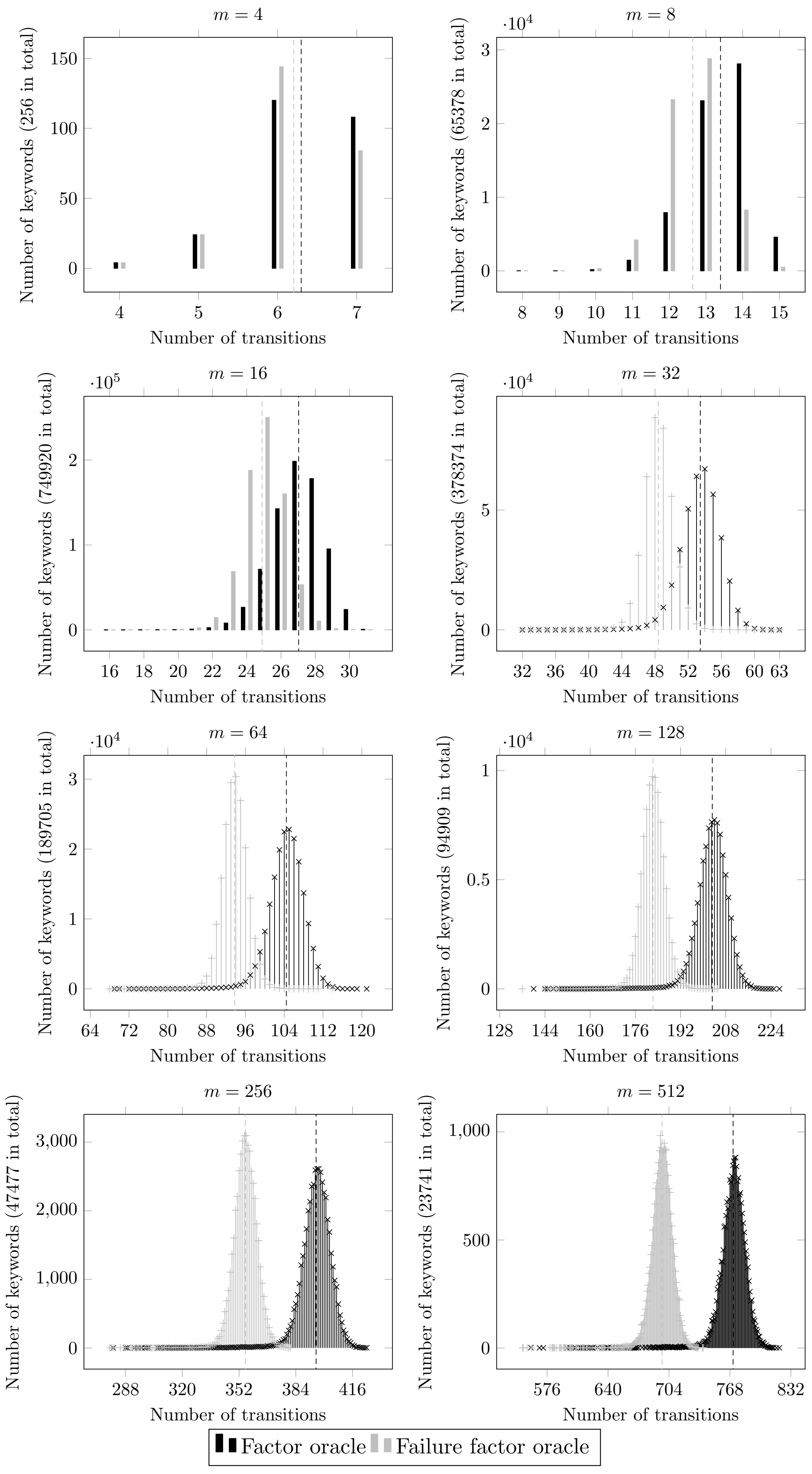

Figure 6: Distribution of number of transitions for the two automaton types, for DNA sequences of lengths $m = 2^n$ for $n = 2, ..., 9$.

Figure 8, and the absence of larger negative values in that figure. The figure also shows that the distribution curves are not single smooth curves, but have relatively more weight towards the left side; this is particularly visible for $m = 64$ and above and presumably due to such (extra) backward or self-symbol transitions becoming necessary. Note however that the data sets for larger m-values are relatively limited compared to the set of possible DNA keywords of such lengths, as also evidenced by the distributions' gaps (for $m = 128$ and above) and outliers (for $m = 64$ and above); thus, it may also be that larger negative size differences than -1 do occur for certain keywords.

Apart from the DNA data set used for the benchmarking reported above, we also benchmarked the FFOs and FOs in terms of size for two other data sets, as reported in [15, p. 184-188]:

- The first set consisted of all generated strings of length m over an alphabet of size m, for values of m in the range of 4..9. (Strings of length < 4 are not considered, as for every string of such length the factor oracle and factor storacle do not differ.) The benchmark results in [15] show that for this data set, FFOs often have one or two fewer transitions than FOs for the same keyword, and FFOs never have more transitions. Furthermore, no backward transitions and hence no cycles are introduced for any of these FFOs.
- The second data set was obtained from [22]:

  > "A list of 109582 English words compiled and corrected in 1991 from lists obtained from the Interociter bulletin board. The original read.me file said that the list came from Public Brand Software. This word list includes inflected forms, such as plural nouns and the -s, -ed and -ing forms of verbs."

  As for the set of generated strings, words of length < 4 were ignored. Figure 9 shows the distribution of the set (including words of length < 4)[3]. As reported in [15, p. 188], "Comparing failure factor oracles to factor oracles, the savings are 1.07% on average for length 5, 4.932% for length 9, and 8.913% for length 15" and "the percentage of savings increases with increasing word length." As with the set of generated strings, the FFO typically saves one or a few transitions over the corresponding FO, and never has more transitions than the FO.

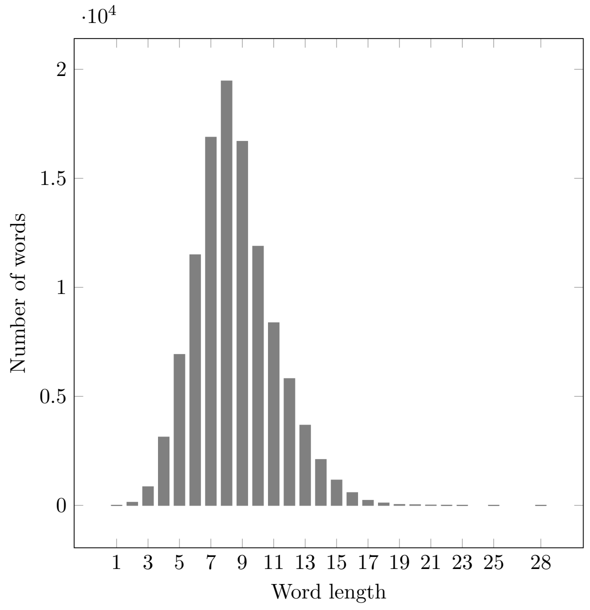

Figure 9: Distribution of lengths of the list of English words. No words of lengths 24, $26 - 27$ or of length over 28 occur. Words of length < 4 were not used for benchmarking.

6 BACKWARD PATTERN MATCHING USING (FAILURE) FACTOR ORACLES

Factor oracles are used in what has been called the Backward Oracle Matching algorithm, first described in [9]. That algorithm is part of a large group of efficient keyword pattern matching algorithms, including the Boyer-Moore and Commentz-Walter algorithms [23, 24, 25, 26]. All of these algorithms share a common approach: they move through the text to be searched from left to right, and at a given position try to match backward, that is from right to left, in order to ascertain whether the keyword appears, and if not, how far to move forward in the text, depending on what has been read.

In [7, 18, 19, 27], Cleophas, Watson and Zwaan report on different versions of a taxonomy of (single and multiple) keyword pattern matching algorithms, with a major branch devoted to such backward matching algorithms. Such a taxonomy classifies algorithms, and is typically depicted as a graph, with nodes corresponding to algorithms (ranging from abstract to concrete), and branches to the addition of an algorithm detail to obtain a new algorithm, which is shown to be a correct refinement or extension of its parent algorithm. The root node of the taxonomy corresponds to a highly abstract algorithm: essentially a precondition and postcondition defining the keyword pattern matching problem, with a symbolic statement S in between to establish the postcondition given the precondition. With the addition of details in order, more and more concrete algorithms are added, leading to either previously published algorithms or to completely new ones.

The taxonomy graph is depicted in Figure 10, taken from [7, 18]. Cleophas et al. [7, 18] describe the taxonomy, including all details, and show how the (single keyword) Backward Oracle Matching algorithm can be classified as part of the taxonomy. The algorithm is identified by detail sequence (P+, S+, GS=FO, EGC=RFO, SSD, NFS, ONE, OKW):

- OKW at the end of the sequence indicates that the algorithm solves the keyword pattern matching problem for the case of <u>O</u>ne <u>K</u>ey<u>w</u>ord.

[3] Taken from [15] with correction, as the original incorrectly excluded the single word of length 25 in the data set.

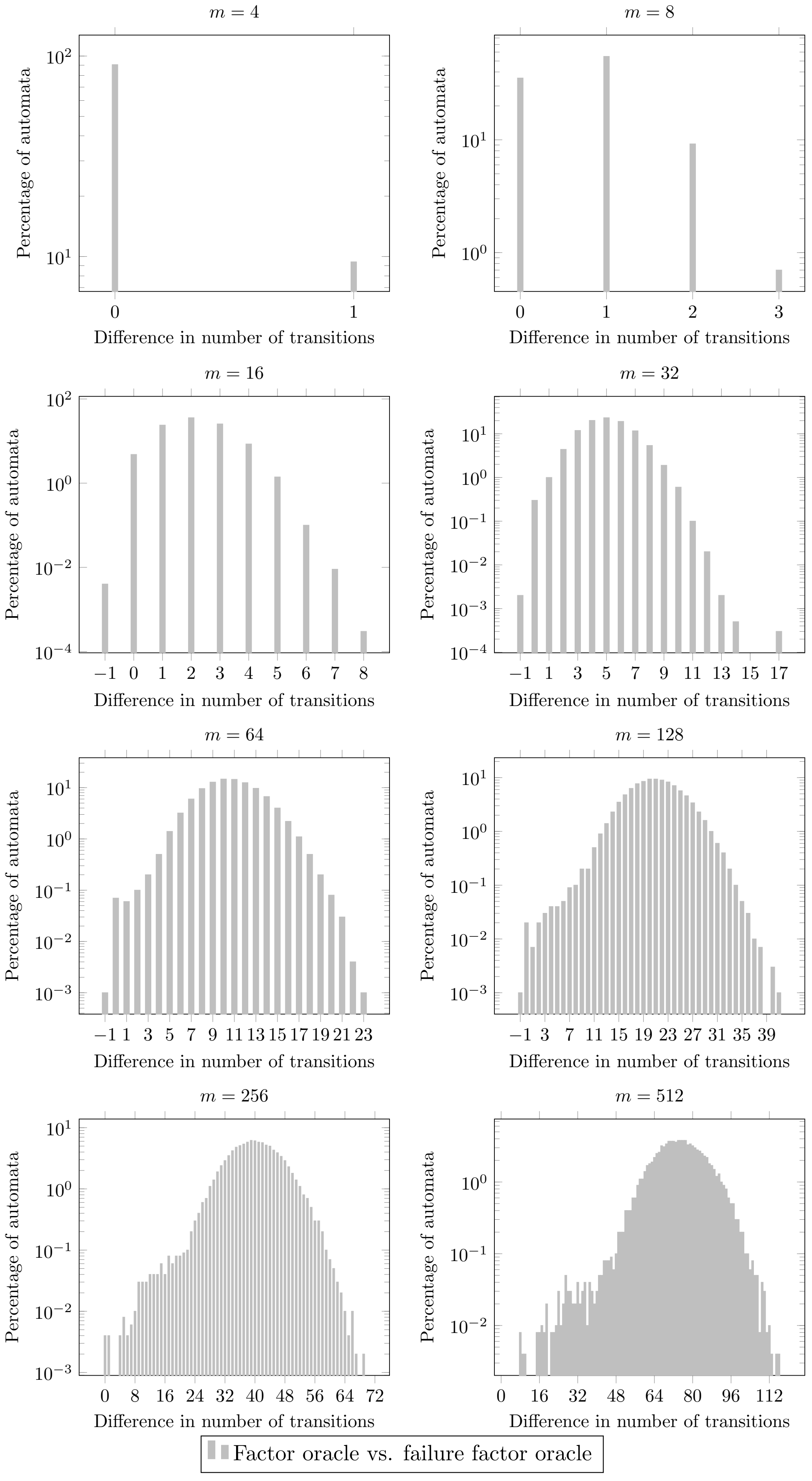

Figure 8: Distribution of difference in number of transitions for factor oracle versus failure factor oracle, for DNA sequences of lengths $m = 2^n$ for $n = 2, ..., 9$.

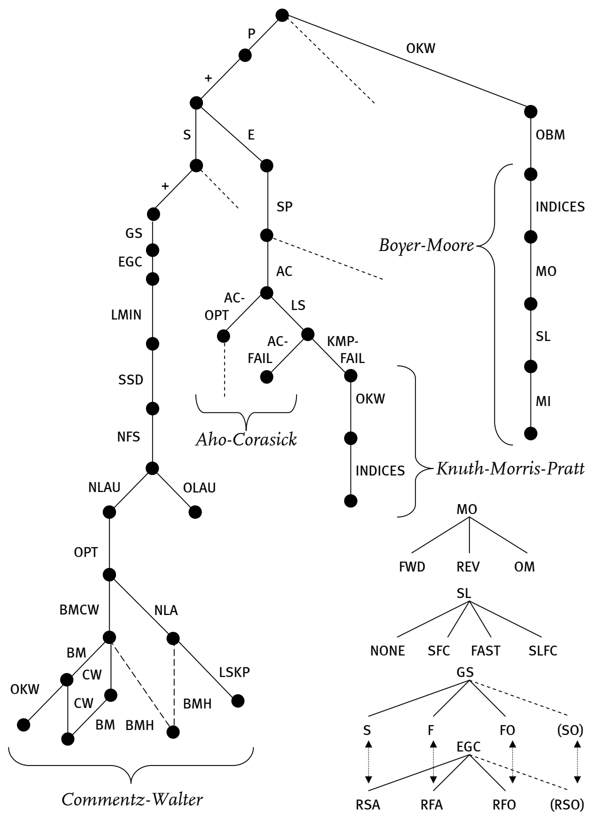

Figure 10: ([7, 18]) A taxonomy of keyword pattern matching algorithms.

- P+ indicates that the algorithm processes the text by moving through it from left to right, i.e. it considers prefixes of the text of increasing length.
- S+ indicates that given a position in the text, a match attempt is made by reading backward, i.e. given a prefix of the text, suffixes of this prefix are considered in increasing length.
- GS=FO refers to Guard Strengthening, by only reading backwards from a given position as long as what is read is part of the Factor Oracle language for the (reversed) keyword(s), and EGC=RFO refers to the use of a Reverse Factor Oracle for Efficient Guard Computation.
- SSD indicates the use of a Safe Shift Distance upon a mismatch, i.e. whenever what has been read backwards is not in the factor oracle language for the keyword(s). A shift distance is *safe* if it is guaranteed to shift forward by *at least* 1 position, and not to shift past the next occurrence of the keyword(s) in the text.
- The particular shift distance used in Backward Oracle Matching is based on the No-Factor Shift—that is, the shift is based on the knowledge that on a mismatch, the string read backwards using the oracle (say aw) is certainly *not* a factor of the keyword (since it was not recognized by the oracle), while w may be (as it was recognized by the oracle). As a result a shift by $|p| - |w|$ is permitted; since this shift may be 0, the maximum of this NFS and the constant shift distance 1 (indicated by ONE) is used as a safe shift distance.

The use of a failure factor oracle instead of a factor oracle changes the classified Backward Oracle Matching algorithm in small ways. Detail choices GS=FO and EGC=RFO need to be changed to new values, reflecting the use of the FFO instead of the FO; we chose to use GS=FFO and EGC=RFFO, i.e. failure factor oracle instead of factor oracle. Somewhat implicitly, these new detail choices (and hence the use of the FFO) also imply that instead of a DFA with its transition function, an FDFA with its transition function is used.

Assuming the above, (single keyword) Backward Oracle Matching using a Failure Factor Oracle can be represented in the taxonomy as Algorithm (P+, S+, GS=FFO, EGC=RFFO, SSD, NFS, ONE, OKW). Using the algorithmic notation of [7, 18], we show a few of the intermediate algorithms on the path from the taxonomy root to this algorithm, to give the reader a better idea of the taxonomy, the algorithms and derivation steps in it, and how we place the new failure factor oracle variant of Backward Oracle Matching in the taxonomy.

Formally [7, 18] specify the keyword pattern matching problem, given an alphabet V (a non-empty finite set of symbols), an input string/sequence $S \in V^*$, and a finite non-empty pattern set $P = \{p_0, p_1, \ldots p_{|P|-1}\} \subseteq V^*$, as to establish

$$R: \ O = \left(\bigcup l, v, r : lvr = S \wedge v \in P : \{(l, v, r)\} \right),$$

that is to let solution set O be the set of triples (l, v, r) such that l, v and r form a splitting of input string S in three parts: a left context, a keyword occurrence, and a right context. For simplicity, we assume that $P \neq \varnothing$ and that $\varepsilon \notin P$. Here, as in [7, 18], we use the notation for quantifications as introduced by Gries and Schneider in [28, Chapter 8].

For example, for keyword set $P = \{CAT, CGT, GTA\}$ and input string $S = CGTATTCAT$, after algorithm termination set O will contain three triples, i.e.

$$\begin{array}{lll} (\varepsilon, & CGT, & ATTCAT), \\ (C, & GTA, & TTCAT), \\ (CGTATT, & CAT, & \varepsilon). \end{array}$$

By considering prefixes of sequence S in order of increasing length, and considering suffixes of such a prefix in order of increasing length as well, we obtain the following basic algorithm. Operationally, the algorithm moves through text S from left to right, its position being between an initial part u and remaining part r of S, with u's length increasing and r's length decreasing.

From each current position, the algorithm reads from right to left, reading suffixes v of u in order of increasing length. Whenever the currently read v is a keyword, the appropriate corresponding tuple is added to the solution set. The resulting algorithm is Algorithm (P+, S+):[4]

[4]The **as** G → S **sa** statement in this extension of GCL can be read as the if-statement of most mainstream languages, or as **if** $G \to S \ [\!] \ \neg G \to$ **skip fi** in GCL.

```
$u, r := \varepsilon, S$;
$O := \varnothing$;
{ inv $ur = S$
    $\wedge\, O = \left( \bigcup x,y,z : \begin{array}{l} xyz = S \\ \wedge\, xy \leq_p u \\ \wedge\, y \in P \end{array} : \{(x,y,z)\} \right)$ }
do $r \neq \varepsilon \rightarrow$
    $u, r := u(r{\upharpoonright}1), r{\downharpoonright}1$;
    $l, v := u, \varepsilon$;
    { inv $u = lv$ }
    do $l \neq \varepsilon \rightarrow$
        $l, v := l{\downharpoonright}1, (l{\upharpoonright}1)v$;
        as $v \in P \rightarrow O := O \cup \{(l,v,r)\}$ sa
    od
od{ $R$ }
```

To improve the algorithm, we may note that for any sequence w, if $w \notin$ **fact**(P) then any extension of w on the left will not be an element of **fact**(P) either. As a result, the inner repetition can terminate as soon as $(l{\upharpoonright}1)v \notin$ **fact**(P) holds, since then all suffixes of u that are equal to or longer than $(l{\upharpoonright}1)v$ are not in **fact**(P) either and hence not in P.

The inner repetition guard can therefore be strengthened to $l \neq \varepsilon$ **cand** $(l{\upharpoonright}1)v \in$ **fact**(P), which makes $v \in$ **fact**(P) an invariant of the inner repetition as well.

To efficiently determine whether or not $(l{\upharpoonright}1)v \in$ **fact**(P), a factor automaton can be used. Since both FFO and FO recognize a superset of **fact**(P) however, they may be used instead of such a factor automaton, and function **fact** may be replaced by e.g. **factoracle** or **failfactoracle** (the function yielding the language of the FO respectively of the FFO for a keyword) since $(l{\upharpoonright}1)v \notin$ **failfactoracle**(P) (respectively **factoracle**(P)) clearly implies $(l{\upharpoonright}1)v \notin$ **fact**(P). As we will see in the algorithm below, and as mentioned before, in fact the automata are built on P^R, the reverse of the keyword set P, as $(l{\upharpoonright}1)v$ is read in reverse, since the algorithm reads backward from its current position.

When the inner loop terminates, a shift or jump of more than position may be made, as already indicated in the introduction. In the case of Backward Oracle Matching, this shift is based on the value, or more precisely the length, of v. To the basic algorithm presented above, we add the guard strengthening, the introduction of an FFO for efficient guard computation, the safe shift distance, as well as the fact that a single keyword is being considered for matching. This yields Algorithm (P+, S+, GS=FFO, EGC=RFFO, SSD, NFS, ONE, OKW), as shown below, where k represents the *shift function* yielding the safe shift distance, and δ_R the transition function of the FFO for P^R.

```
$u, r := \varepsilon, S$;
$O := \varnothing$;
$l, v := \varepsilon, \varepsilon$;
{ inv $ur = S$
    $\wedge\, O = \left( \bigcup x,y,z : \begin{array}{l} xyz = S \\ \wedge\, xy \leq_p u \\ \wedge\, y = p \end{array} : \{(x,y,z)\} \right)$
    $\wedge\, u = lv \wedge v^R \in$ failfactoracle$(p^R)$
    $\wedge\, \left(l = \varepsilon \text{ cor } ((l{\upharpoonright}1)v)^R \notin \text{failfactoracle}(p^R)\right)$ }
do $r \neq \varepsilon \rightarrow$
    $u, r := u(r{\upharpoonright}k(v)), r{\downharpoonright}k(v)$;
    $l, v := u, \varepsilon$;
    $q := \delta_R(q_0, l{\upharpoonright}1)$;
    { inv $q = \delta_R^*(q_0, ((l{\upharpoonright}1)v)^R)$ }
    do $l \neq \varepsilon$ cand $q \neq \bot \rightarrow$
        $l, v := l{\downharpoonright}1, (l{\upharpoonright}1)v$;
        $q := \delta_R(q, l{\upharpoonright}1)$;
        as $v = p \rightarrow O := O \cup \{(l,v,r)\}$ sa
    od
od{ $R$ }
```

The efficient determination of whether $v = p$ is typically possible because the automaton used is such that if $|v| = |p|$ in the inner loop, this implies $v = p$. This is the case whenever the only string of length $|p|$ recognized by the automaton is p itself.

This holds true for (exact) factor automata, as well as for factor oracles: since they are acyclic, deterministic, and heterogenous, transitions outside the skeleton recognizing p always skip at least one state, hence p is the only string of length $|p|$ recognized. For failure factor oracles, the same reasoning holds true if and only if the FFO has no cycles; if an FFO does, false matches may be detectedterms v of length $|p|$ that do not equal p—and additional vetting of matches might be required if such false matches are unacceptable.

In practice, our extensive benchmarking in the next section never encountered situations where such a false match occurs, i.e. they seem at least very rare. This is plausible, as the (repeated) use of cycles implies the existence of repeated subsequences, without mismatchessomething that is unlikely for DNA, let alone English text.

7 EMPIRICAL RESULTS ON BACKWARD (FAILURE) ORACLE MATCHING

In addition to the results on number of transitions as reported in Section 5, we also performed runtime experiments with the constructed FOs and FFOs.

As a first step, we used the automata for basic sequence processing: whenever the automaton gets stuck, we assume there to be a symbol transition to state 0, i.e. we reset the automaton to state 0 and continue processing from the next symbol in the sequence onward. This initial rudimentary processing was performed using FFOs constructed as per Section 5 from the DNA sequences of the various chromosomes of *Saccharomyces cerevisiae (strain S288c)* as available at [21]. The sequence to process using each of these

FFOs was obtained by taking the first 1000000 characters of the 12156677 long concatenated sequence from Section 5.

Initial time measures when using FFOs instead of FOs were quite large, with FFOs using multiples of the times measured for FOs. A number of reasons contribute to these long running times:

- If we assume character frequency to be the same for all four DNA characters, on average in 75% of cases a failure transition lookup will be *attempted* when processing a symbol (but the lookup might not succeed, as it may well be that no failure transition exists from the given state).
- In addition, ignoring the small chance that external transitions added are symbol transitions (see above—less than 1% of the external transitions), the ratio of average number of transitions to the keyword length m ranges from 155% (for $m = 4$) to 136% (for $m = 512$) of m (cf. the dotted lines for the average in the respective graphs). Since failure transitions form a function, this means that *on average* $36 - 55\%$ of states has a failure transition leading from it; hence there is a fairly high chance that for every symbol to be processed, at least one but possibly multiple failure transitions in sequence will be processed.
- In particular, unless a keyword is of the form c^m, state 0 will have a failure transition, so after every reset of the automaton, there's a 75% chance that that failure transition will be used.

Our initial attempts to improve on the running times when using FFOs therefore implemented partial memoization or caching of up to two states and one transition label: state 0, the symbol labelling the unique symbol transition from state 0 (leading to state 1), and the state to which the failure transition from state 0 leads (if it exists) are all cached. Thus, whenever the automaton gets stuck, we can first lookahead at the next symbol to be processed:

- If the lookahead symbol equals the cached symbol labelling the transition from state 0 to state 1, the automaton is moved to cached state 1 directly and processing can continue with the sequence position directly following the lookahead symbol position.
- Otherwise, if the failure transition from state 0 exists and hence is cached, the automaton directly moves to the cached destination state of the failure transition, and processing continues at the lookahead symbol position in the sequence.
- Otherwise, if the failure transition from state 0 does not exist, the automaton is moved to the cached state 0 and processing continues at the lookahead symbol position in the sequence (which in the next iteration of the processing will cause the automaton to get stuck, as neither a matching symbol transition nor a failure transition exists).

This partial memoization or caching approach substantially reduced processing time in our preliminary runtime experiments, with results on small data sets with limited numbers of runs showing that FFO processing took $34 - 88\%$ more time than FO processing, depending on keyword length.

Following these initial experiments, we implemented and benchmarked true pattern matching algorithms as well: both the Backward Oracle Matching algorithm using FOs as well as the version using FFOs, as described in Section 6, were implemented and benchmarked. In the implementation of the version using FFOs, the caching techniques described above were applied.

The two pattern matching algorithms were benchmarked on two data sets: the FOs and FFOS constructed for the DNA data set discussed in Section 5, as well as the ones constructed for the English natural language data set discussed there and in [15].

For the DNA data set, the single sequence to match against was the same 10^6 character sequence used in that Section. For the English data set, the single text to match agains corresponded to the first 10^6 characters of the raw file of Project Gutenberg's EBook of Webster's Unabridged Dictionary [29].

For both data sets, various subsets of the data sets were used for benchmarking; depending on the subset chosen, the results were different, which is as expected: depending on the particular keyword to be matched against, different behaviour in terms of transitions utilised during match attempts is expected. Nevertheless, the various benchmark results show that the use of FFO-based Backward Oracle Matching consistently brings a penalty over the use of FO-based Backward Oracle Matching. Depending on the subsets selected, i.e. the keywords used, the time penalties ranged from ca. 30% to over 100%—despite the use of partial memoization / caching techniques for the FFO-based algorithm (as discussed above).

Although the above benchmarking is limited in terms of texts processed (just a single one) and sets of keywords used to match, the results are such that we believe their gist to hold true more generally: in most cases, the use of failure factor oracles leads to substantially increased running times compared to the use of factor oracles. The case of DNA sequences of length 4 forms the exception. It can be explained by the fact that the FO version of the algorithm was not enhanced with caching similar to that applied for the FFO version. For this particular keyword length (4) and alphabet size (4), the FFO version slightly outperforms the FO version as a result. For longer keywords, and for the larger alphabet size of the English language, this is not the case, as the number of failure transitions used during processing increases to a level where the caching no longer offsets this effect.

8 CONCLUSIONS AND FUTURE WORK

We have presented an improved construction algorithm for directly constructing a failure factor oracle, an algorithm that is suitable for sequences of any length, including those typically seen in the context of DNA processing. This direct FFO construction does not necessarily result in an FDFA that is language equivalent to the corresponding FO, but it does result in an automaton recognizing at least all factors of a given sequence. We empirically compared the resulting failure and traditional (non-failure) factor oracle automata on number of transitions, using DNA sequences with lengths in the range $4 - 512$. The results show that the use of the improved FFO construction saves up to 10% in *number of* transitions—similar to the savings shown in [15] for generated as well as English dictionary data sets. In addition to these savings of up to 10% in number of transitions, it should be noted that on average $36 - 55\%$ (depending on keyword length) of an FFO's transitions are failure transitions, which are more compactly representable than symbol transitions. Altogether this resulted in promising savings, given the often large amounts of sequence data typically occurring in DNA processing applications (as well as natural language ones), particularly for the use of oracles as indexes for (repeated) factor detection.

Processing time of FFOs for basic sequence processing in our first empirical evaluations was drastically worse than that of corresponding FOs, although the use of partial memoization showed promise in substantially reducing this overhead. We also discussed FO-based Backward Oracle Matching as well as its FFO-based variant. Empirical evaluation of both algorithms showed that even for this more realistic application to pattern matching, and while employing the partial memoization approach for the FFO-based variant, the latter had considerable overhead compared to the FO-based algorithm. The results may thus be called disappointing. As future work, it might be worthwhile to consider more intelligent introduction of failure transitions, e.g. not introducing a failure transition from the start state, since such transitions, while saving lots of symbol transitions, are used far too often; or more generally, introduce failure transitions only there, where they can be expected not to be frequently used. It is very well conceivable however that such approaches are better applied ex post facto to a classical factor oracle, i.e. in the course of applying one of the DFA-to-FDFA algorithms described in [13, 14].

REFERENCES

[1] A. Lefebvre, T. Lecroq, H. Dauchel and J. Alexandre. "FORRepeats: detects repeats on entire chromosomes and between genomes". *Bioinformatics*, vol. 19, no. 3, pp. 319–326, 2003.

[2] C. S. Iliopoulos, D. G. Kourie, L. Mouchard, T. K. Musombuka, S. P. Pissis and C. de Ridder. "An algorithm for mapping short reads to a dynamically changing genomic sequence". *J. Discrete Algorithms*, vol. 10, pp. 15–22, 2012.

[3] C. de Ridder, D. G. Kourie and B. W. Watson. "FireµSat: An algorithm to detect microsatellites in DNA". In J. Holub and J. Zdárek (editors), *Stringology*, pp. 137–150. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2006. ISBN 80-01-03533-6.

[4] R. Kato. "Finding Maximal Repeats with Factor Oracles". Tech. Rep. TR-C190, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2004.

[5] S. Faro and T. Lecroq. "The exact online string matching problem: A review of the most recent results". *ACM Comput. Surv.*, vol. 45, no. 2, p. 13, 2013.

[6] M. Crochemore and W. Rytter. *Jewels of Stringology - Text Algorithms*. World Scientific Publishing, 2003.

[7] L. Cleophas, B. W. Watson and G. Zwaan. "A new taxonomy of sublinear right-to-left scanning keyword pattern matching algorithms". *Sci. Comput. Program.*, vol. 75, no. 11, pp. 1095–1112, 2010.

[8] G. Navarro and M. Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[9] C. Allauzen, M. Crochemore and M. Raffinot. "Efficient Experimental String Matching by Weak Factor Recognition". In *Proceedings of the 12th conference on Combinatorial Pattern Matching*, vol. 2089 of *LNCS*, pp. 51–72. 2001.

[10] A. V. Aho and M. J. Corasick. "Efficient string matching: an aid to bibliographic search". *Communications of the ACM*, vol. 18, pp. 333–340, 1975.

[11] D. E. Knuth, J. H. Morris and V. R. Pratt. "Fast pattern matching in strings". *SIAM Journal of Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[12] M. Crochemore and C. Hancart. "Automata for Matching Patterns". In G. Rozenberg and A. Salomaa (editors), *Handbook of Formal Languages*, vol. 2. Springer, 1997.

[13] D. G. Kourie, B. W. Watson, L. Cleophas and F. Venter. "Failure Deterministic Finite Automata". In *Proceedings of the Prague Stringology Conference 2012*. Department of Theoretical Computer Science, Czech Technical University, Prague, September 2012.

[14] H. Björklund, J. Björklund and N. Zechner. "Compact representation of finite automata with failure transitions". Tech. Rep. UMINF 13.11, Umeå University, 2013.

[15] L. Cleophas, D. G. Kourie and B. W. Watson. "Weak Factor Automata: Comparing (Failure) Oracles and Storacles". In *Proceedings of the Prague Stringology Conference, Prague, Czech Republic, September 2-4, 2013*. September 2013.

[16] L. Cleophas, G. Zwaan and B. W. Watson. "Constructing Factor Oracles". *Journal of Automata, Languages and Combinatorics*, vol. 10, no. 5/6, pp. 627–640, 2005.

[17] L. Cleophas and B. W. Watson. "On Factor Storacles: an Alternative to Factor Oracles?" In *Festschrift for Bořivoj Melichar*. Department of Theoretical Computer Science, Czech Technical University, Prague, August 2012.

[18] L. Cleophas, B. W. Watson and G. Zwaan. "A new taxonomy of sublinear keyword pattern matching algorithms". Tech. Rep. 04/07, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, March 2004.

[19] B. W. Watson and G. Zwaan. "A taxonomy of sublinear multiple keyword pattern matching algorithms". *Science of Computer Programming*, vol. 27, no. 2, pp. 85–118, 1996.

[20] L. Cleophas, D. G. Kourie and B. W. Watson. "Efficient representation of DNA data for pattern recognition using failure factor oracles". In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference (SAICSIT) 2013*, pp. 369–377. 2013.

[21] `http://www.dna-algo.co.za/downloads.htm`.

[22] "English wordlist". `http://www.sil.org/linguistics/wordlists/english`.

[23] R. S. Boyer and J. S. Moore. "A fast string searching algorithm". *Communications of the ACM*, vol. 20, no. 10, pp. 62–72, 1977.

[24] W. Rytter. "A Correct Preprocessing Algorithm for Boyer-Moore String-Searching". *SIAM Journal of Computing*, vol. 9, no. 3, pp. 509–512, 1980.

[25] B. Commentz-Walter. "A string matching algorithm fast on the average". In H. A. Maurer (editor), *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*, pp. 118–132. Springer, Berlin, 1979.

[26] B. Commentz-Walter. "A string matching algorithm fast on the average". Tech. Rep. TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.

[27] B. W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* Ph.D. thesis, Faculty of Computing Science, Technische Universiteit Eindhoven, September 1995.

[28] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math.* Springer, New York, NY, 1993.

[29] "Project Gutenberg EBook of Webster's Unabridged Dictionary", August 2009. EBook no. 29765. `http://www.gutenberg.org/ebooks/29765`.