Weak Factor Automata: Comparing (Failure) Oracles and Storacles

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

[1] FASTAR Research Group, Department of Computer Science, University of Pretoria, Private Bag X20, 0028 Hatfield, Pretoria, Republic of South Africa
{loek,derrick}@fastar.org
[2] FASTAR Research Group, Department of Information Science, Stellenbosch University, Private Bag X1, 7602 Matieland, Republic of South Africa
bruce@fastar.org
http://www.fastar.org

Abstract. The *factor oracle* [3] is a data structure for weak factor recognition. It is a deterministic finite automaton (DFA) built on a string p of length m that is acyclic, recognizes at least all factors of p, has $m+1$ states which are all final, is homogeneous, and has m to $2m-1$ transitions. The *factor storacle* [6] is an alternative automaton that satisfies the same properties, except that its number of transitions may be larger than $2m-1$, although it is conjectured to be linear with an upper bound of at most $3m$. In [14] (among others), we described the concept of a *failure automaton* i.e. a *failure DFA* (FDFA), in which so-called failure transitions are used to reduce the total number of transitions and thus reduce representation space compared to the use of a DFA. We modify factor oracle and storacle construction algorithms to introduce failure arcs *during* the respective automata's construction. We thus end up with four deterministic automata types for weak factor recognition: factor oracle, factor storacle, failure factor oracle, and failure factor storacle. We compare them empirically in terms of size. The results show that despite the relative simplicity of (failure) factor (st)oracles, the failure versions show additional savings of 2–7% in number of transitions, for generated keywords of length 5–9, and of e.g. 5–9% for English words of lengths around 9–15. This may already be substantial in memory-restricted settings such as hardware implementations of automata. The results indicate the gains increase for longer keywords, which seems promising for applications in DNA processing and intrusion detection. Furthermore, our results provide a rather negative result on storacles: apart from rare cases, factor storacles do not have fewer transitions than factor oracles, and similarly for failure factor storacles versus failure factor oracles.

Keywords: factor oracle, approximate automaton, failure automaton, weak factor recognition, pattern matching

1 Introduction

The *factor oracle* is a data structure for weak factor recognition. It is an automaton built from a string p of length m that (a) is acyclic, (b) recognizes at least all factors of p, (c) has $m+1$ states (which are all final), and (d) has m (at least, one for each letter in p) to $2m-1$ transitions (cf. [3]). In addition, (e) the resulting automaton is homogeneous, i.e. for every state, all of its incoming transitions are on the same symbol. An example factor oracle is given in Figure 1. Factor oracles are introduced in [3] as an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. In such algorithms, a window on a text is read backward while attempting to match a keyword factor. When this fails, the window is shifted using the information of the longest factor matched and the mismatching character.

Loek Cleophas, Derrick G. Kourie, Bruce W. Watson: Weak Factor Automata: Comparing (Failure) Oracles and Storacles, pp. 176–190.
Proceedings of PSC 2013, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05330-0 © Czech Technical University in Prague, Czech Republic

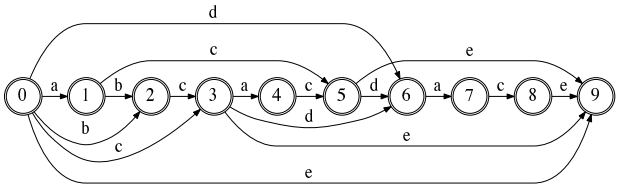


Figure 1: Factor oracle (with initial state 0) recognizing a superset of **fact**(p) (including for example *cace* $\notin$ **fact**(p)), for $p = abcacdace$. The automaton has 17 transitions.

Instead of an automaton recognizing exactly all factors of the keyword, it is possible to use a factor oracle: although it recognizes more strings than just the factors and thus might read backwards longer than necessary, it cannot miss any matches. The advantage of using factor oracles is that they are easier to construct and take less space to represent compared to automata that were previously used in these factor-based algorithms, such as suffix, factor and subsequence automata. This is due to the latter automata lacking one or more of the essential properties of the factor oracle.

In [7], we presented an alternative construction algorithm for factor oracles. This algorithm was based on considering the suffixes of the string p in order of decreasing length. While being $\mathcal{O}(m^2)$ and not linear like the algorithm in [3], this construction is easier to understand. (It also makes some of the factor oracle's properties immediately obvious, while making some others harder to prove.) An extended version of [7] appears as [8] and in the Master's thesis [10, Chapter 4]. In those versions, some properties of the language of a factor oracle are discussed as well. The thesis also discusses the implementation of the factor oracle in the SPARE Time toolkit. A further extended and revised version of the work appears in [9]. The language of a factor oracle was finally characterized completely in a paper by Mancheron and Moan [15]. Related to the factor oracle, the *suffix oracle*—in which only those states corresponding to a suffix of p are marked final—is introduced in [3]. In [5], the authors present a statistical average-case analysis on the size of factor and suffix oracles.

In [6] we presented the *factor storacle*, short for *shortest forward transition factor oracle*. The factor storacle is an alternative automaton that satisfies the same properties as the factor oracle does, except property (d) mentioned earlier: in contrast to the case of the factor oracle for the same keyword, the factor storacle's number of transitions may be larger than $2m-1$, although it is conjectured to be linear with an upper bound of at most $3m$. We presented a construction algorithm for factor storacles as well as a limited empirical comparison of factor oracles and factor storacles, showing the maximum numbers of transitions the factor oracle and factor storacle for particular string lengths may have, and leading to the conjecture mentioned above. For certain keywords, the factor storacle has a *smaller* number of transitions than the factor oracle, although such cases turn out to be rare, as we empirically show in this paper. Figure 2 shows an example factor storacle (having one less transition than the corresponding factor oracle depicted in Figure 1).

In [14], we described the concept of a *failure automaton* i.e. a *failure DFA* (FDFA). In such an automaton so-called failure transitions are used to reduce the total number of transitions compared to a DFA for the same language. This is done to reduce the space needed to represent the automaton compared to the space usage of a DFA representation. Björklund et al. in [4] recently showed that even without changing the

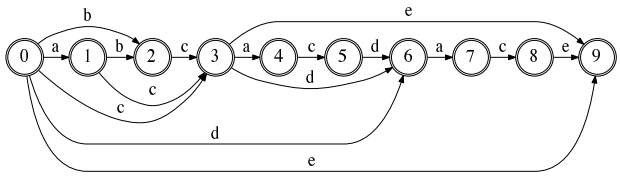


Figure 2: Factor storacle (with initial state 0) recognizing a superset of $\mathbf{fact}(p)$ (including e.g. $abce \notin \mathbf{fact}(p)$), for $p = abcacdace$. The automaton has 16 transitions.

state set from DFA to FDFA, the problem of minimizing the number of transitions by replacing symbol transitions by failure transitions is unfortunately NP-complete, although it can be approximated efficiently within a factor of $\frac{2}{3}$.

The concepts of factor oracle and factor storacle serve to reduce memory usage compared to a factor automaton, while the concept of an FDFA does the same for the general DFA case, albeit in different ways. It is therefore of interest to combine the basic ideas and empirically investigate the results. In the current paper, we thus combine the ideas of factor oracle and storacle on the one hand, and failure automata on the other hand. We modify the factor oracle and factor storacle construction algorithms to introduce failure arcs *during* the respective automata's construction. We thus end up with four kinds of deterministic automata for weak factor recognition: the factor oracle, factor storacle, failure factor oracle, and failure factor storacle. We compare them empirically in terms of size, using both randomly generated keywords as well as English dictionary keywords for the construction process.

After discussing preliminaries, we consider suffix-based factor oracle and factor storacle construction in Section 2. We present our previously existing construction algorithms for these two cases. In Section 3 we present our modified algorithms, directly constructing the failure factor oracle and the failure factor storacle respectively, and we discuss the properties of these two automata types. Section 4 presents and analyses our preliminary benchmarking results in comparing the four resulting automata types; these results focus on size of the resulting automata in terms of number of (symbol and failure) transitions. Section 5 provides concluding remarks as well as a discussion of ideas for future research in this subject area.

2 Suffix-based Construction of the Factor Oracle and Factor Storacle

Formally, a *string* $p = p_1 \cdots p_m$ 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.

In Algorithm 1 the factor oracle construction algorithm given in [7,9] is repeated. In steps 1 to 4 the algorithm constructs a 'skeleton' automaton for p—recognizing $\mathbf{pref}(p)$. In steps 5 to 8, it then considers, in decreasing order of length, each proper suffix $p_i \cdots p_m$ of p. For each such suffix, it determines the longest prefix recognised by the automaton to date—i.e. the longest path starting from state 0 and ending in some state j that spells out $p_i \cdots p_k$ $(i - 1 \leq k \leq m)$. If such a suffix $p_i \cdots p_m$ is already recognized (i.e. if $k = m$), then no transition needs to be constructed. If on the other

hand the complete suffix is not yet recognized—i.e. if $p_i \cdots p_k$ is the longest prefix recognised where $k < m$) and if the recognition path ends at state j—then a transition is inserted from state j to state $k+1$. It can be easily shown that the language recognised by the resulting automaton is a superset of $\mathbf{pref}(\mathbf{suff}(p)) = \mathbf{fact}(p)$.

Algorithm 1 Build_Oracle($p = p_1 p_2 \cdots p_m$)

```
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 ≤ k ≤ m)
7:     if k ≠ m then
8:         Build a new transition from j to k + 1 on symbol p_{k+1}
```

This algorithm is $\mathcal{O}(m^2)$. The factor oracle on p built using this algorithm is referred to as Oracle(p) and the language recognized by it as $\mathbf{factoracle}(p)$.

Our factor storacle construction algorithm, presented in [6], is similar to our factor oracle construction algorithm. It is reproduced in Algorithm 2. It also constructs a 'skeleton' automaton for p—recognizing $\mathbf{pref}(p)$—and then also constructs a path for each of the proper 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 prefix of such a suffix that is recognized.

A transition on the next, non-recognized symbol is then created, from the state in which this longest prefix of the suffix is recognized. Instead of creating such a transition to the unique state from which the remainder of that suffix is known to be recognized, as is done in the factor oracle construction above, this transition is constructed to go to the next state from the current state onward that has an incoming transition on the non-recognized symbol. That is, the factor storacle construction algorithm in such a case constructs the shortest forward transition that keeps the automaton homogeneous. This procedure of creating transitions is repeated while the complete suffix is not yet recognized.

Algorithm 2 Build_Storacle($p = p_1 p_2 \cdots p_m$)

```
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 ≤ k ≤ m)
7:     while k ≠ m do
8:         Let the first state from state j onward that has an incoming transition on p_{k+1} be state l
           (j < l <= k + 1)
9:         Build a new transition from j to l on symbol p_l (= p_{k+1})
10:        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 ≤ k ≤ m)
```

This algorithm is $\mathcal{O}(m^3)$, although that is a coarse upper bound. The factor storacle on p built using this algorithm is referred to as Storacle(p) and the language recognized by it as **factstoracle**(p).

As stated in [6], the difference between this algorithm and the $\mathcal{O}(m^2)$ factor oracle construction algorithm originates from the choice of the target of the first (if any) newly created transition for each proper suffix:

- In this algorithm, that newly created transition leads to the next state (from a particular state onward) that has an incoming transition on the non-recognized symbol. This procedure may then need to be repeated for further symbols of the suffix to be recognized.
- In the case of the factor oracle construction, the newly created transition leads to the unique state from which the remainder of the suffix leads to the last state of the automaton—thus immediately guaranteeing that the entire suffix is recognized.

We summarize the most important properties of factor oracles and factor storacles. All of these were known before; some proofs are therefore omitted or sketched, and can be found in e.g. [3,7,9,6]. The first properties mentioned correspond to properties (a)–(c) and (e) from the introduction, and hold for factor oracles and factor storacles.

Property 1. Oracle(p) and Storacle(p) are acyclic automata.
Proof idea: For the factor oracle, it is obvious that transitions created are always forward ones; for the factor storacle, it can be shown that the transitions created may be different from those created for the factor oracle, but are still forward ones.

Property 2. **fact**(p) $\subseteq$ **factoracle**(p) and **fact**(p) $\subseteq$ **factstoracle**(p).

Property 3. For p of length m, Oracle(p) and Storacle(p) each have exactly $m + 1$ states.

Property 4 (Homogeneousness). All transitions reaching a state i of Oracle(p) and Storacle(p) are labeled by p_i.

Furthermore, factor oracles and factor storacles satisfy the following obvious property:

Property 5 (Weak determinism). For each state of Storacle(p) or Oracle(p), no two outgoing transitions of the state are labeled by the same symbol.

As stated before, property (d), the remaining property enumerated in the introduction, only holds for factor oracles, while a weaker property holds for factor storacles.

Property 6. For p of length m, Oracle(p) has between m and $2m - 1$ transitions.

Since the factor storacle construction algorithm we presented might create multiple transitions per proper suffix of the keyword, this property does not hold for factor storacles. [6] showed the following very coarse upper bound on the total number of transitions of the factor storacle:

Property 7. For p of length m, Storacle(p) has between m and $m(m+1)/2$ transitions.
Proof: The lower bound follows from the second **for**-loop of the algorithm. Disregarding any properties of the keyword and alphabet used (except for the keyword's length

m), an upper bound of $m(m+1)/2$ can be proven in at least two ways. Firstly, the sum of the lengths of all the suffixes of a keyword of length m, including the keyword itself, equals $m(m+1)/2$. Secondly, since all transitions are forward transitions, and the factor storacle is kept homogeneous, there can be at most one transition between each pair of states, hence at most $(|Q|-1)|Q|/2$ in total, and this equals $m(m+1)/2$ since $m = |Q| - 1$. □

[6] also conjectured a linear upper bound on the number of transitions of the factor oracle, based on empirical evidence; experiments generating all keywords of length m out of an alphabet of size $|m|$ (modulo renaming of alphabet symbols) showed that the upper bound increases from at most $2m$ for lengths up to $m = 7$ to $2m + 5$ for length $m = 12$, i.e. grows linearly in the range of the experiments.

Conjecture 8. For p of length m, Storacle(p) has a linear number of transitions, bounded above by $3m$.

3 Suffix-based Construction of the Failure Factor Oracle and Failure Factor Storacle

In [14], we give a general algorithm for constructing a *failure deterministic finite automaton* (FDFA) based on a given deterministic finite automaton (DFA). The algorithm ensures that the constructed FDFA is language-equivalent to the given DFA. Such an FDFA in essence forms a generalization of the failure function Aho-Corasick automaton [2,17]: from a finite set of keywords (as in normal Aho-Corasick), to the general/arbitrary regular language case. In essence, an FDFA is a DFA, but may have so-called failure transitions apart from normal symbol transitions. Such transitions are introduced to save space: under certain conditions, a single failure transition can be used as default instead of multiple symbol transitions. These failure transitions are represented by the function $\mathfrak{f}$ in the definition below.

Definition 9 (FDFA [14]). *$\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ 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.*

As with a DFA, a simple string recognition algorithm can be used to determine whether or not a given string is part of the FDFA's language. The algorithm corresponds to that of a DFA by consuming an input symbol and moving to a next state if there is an out-transition from the current state on the current input symbol. However, if there is no such out-transition, but a failure transition, then the failure transition determines the new state, but the current input symbol is not consumed.

The above FDFA definition may lead to complications in the presence of certain types of cycles in the failure function. More precisely, cycles in which, for one or more symbols, no state in the cycle has an out-transition labeled by this symbol are problematic for the associated FDFA string recognition algorithm. [14] called these *divergent failure cycles*, and ensured that the FDFA construction algorithm presented simply does not create such divergent failure cycles. In the present setting, no failure cycles are created at all, circumventing the potential complications altogether.

In [14], FDFAs were created by taking (complete) DFAs and transforming them. Here, we introduce failure transitions *during* construction of weak factor automata. We do so by slight modifications of the factor oracle and factor storacle construction algorithms presented before. These modifications lead to construction algorithms for

what we call the *failure factor oracle* and the *failure factor storacle* respectively. It should be noted that the resulting automata are not necessarily language-equivalent to the original factor oracle or factor storacle respectively, but we are not concerned with such language-equivalence here: what matters is that the resulting automata recognize at least all factors of the given keyword.

Algorithm 3, the failure factor oracle construction algorithm, is similar to the factor oracle one, Algorithm 1. The main differences are in lines 6 and 8–11: in line 6, from state j processing continues with (0 or more) existing failure transitions leading to a state j', to prevent constructing a failure transition (in line 9) from a state that already has an outgoing failure transition. (Note that the recognition path that leads to state j may also contain failure transitions—another implicit difference to Algorithm 1.) In line 9, in case $k > j'$, instead of a symbol transition from j to $k+1$ on symbol p_{k+1}, a failure transition from j' to k is constructed. Note that a transition on symbol p_{k+1} from state k to state $k+1$ will exist, due to lines 1–4 of the algorithm, and hence processing of $p_i \cdots p_k p_{k+1}$ will have the automaton end up in state $k+1$, just as it would in the original factor oracle.

Our initial version of the algorithm did not have the inner **if**-statement, assuming $k > j'$ to always hold inside the outer **if**-statement, and therefore always building a new failure transition from j' to k, keeping the automaton acyclic. For the large data sets we used in the experiments reported further on in this paper, this holds true, but it is not true in general: with increasing keyword length, it becomes possible in rare cases for $k > j'$ not to hold. In some such cases, cycles of failure transitions arise, which in some cases lead to divergent failure cycles and even live-lock of the construction algorithm. The **else**-case of lines 10–11 ensures that this does not happen, by creating an appropriate non-forward *symbol* transition instead of a non-forward failure transition. The failure factor oracle in general thus does not have the acyclicity property of the factor oracle, but our initial experiments with sets of longer keywords (on a DNA alphabet) show such cases to be rare ($< 0.001\%$ of 749920 keywords tested of length 16 rising to ca. 1% of 5935 keywords tested of length 1024).

Algorithm 3 Build_Failure_Oracle($p = p_1 p_2 \cdots p_m$)

```
 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 ≤ k ≤ m), and let the longest failure transition path from j end in state j'
 7:     if k ≠ 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
```

This algorithm is $\mathcal{O}(m^2)$. The failure factor oracle on p built using this algorithm is referred to as FailureOracle(p). It is easy to show that, apart from acyclicity, the properties of the factor oracle mentioned previously do hold for the failure version.

The failure factor storacle construction algorithm, Algorithm 4, is similar to the factor storacle construction algorithm, Algorithm 2. The main differences are in lines 6 and 8–13: on lines 6 and 13, as for the failure factor oracle construction above,

processing continues with existing failure transitions, to prevent constructing one from a state that already has an outgoing failure transition; on lines 9 and 10, instead of a symbol transition from j to l on symbol p_{k+1}, a failure transition from j' to $l-1$ ($j' < l <= k+1$) is constructed. Note that a path (possibly using failure transitions) to process symbol p_{k+1} from that state l may not exist, and processing of the current suffix thus has to continue, as in Algorithm 2. As with Algorithm 3, an **else**-case is added to prevent divergent failure cycles from arising in case $k \leq j'$.

Algorithm 4 Build_Failure_Storacle($p = p_1 p_2 \cdots p_m$)

```
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 ≤ k ≤ m), and let the longest failure transition path from j end in state j'
7:     while k ≠ m do
8:         if k > j' then
9:             Let the first state from state j' onward that has an incoming transition on p_{k+1} be state
               l (j' < l <= k + 1)
10:            Build a new failure transition from j' to l − 1
11:        else
12:            Build a new symbol transition on symbol p_{k+1} from j' to k + 1
13:        Let the longest recognized prefix of p_i ··· p_m be recognized in state j and spell out p_i ··· p_k
           (i − 1 ≤ k ≤ m), and let the longest failure transition path from j end in state j'
```

This algorithm is $\mathcal{O}(m^3)$, although that is a coarse upper bound. The failure factor storacle on p built using this algorithm is referred to as FailureStoracle(p).

Figure 3 depicts an example of a failure factor oracle and a failure factor storacle in one: for this particular keyword, the automata happen to be equivalent. Figures 4a and 4b depict the case of keyword *abcaab*, for which the automata differ.

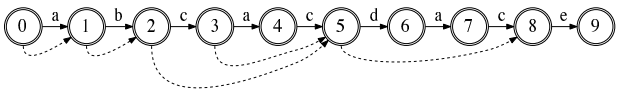
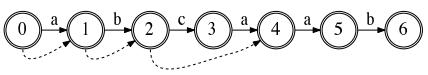
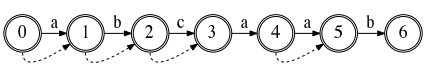


Figure 3: Failure factor oracle and failure factor storacle (with initial state 0) recognizing a superset of **fact**(p) (including for example *cace* $\notin$ **fact**(p), and *acace* not recognized by Oracle(p)), for $p = abcacdace$. The automaton has 14 transitions.



(a) Failure factor oracle (with initial state 0). The automaton has 9 transitions.



(b) Failure factor storacle (with initial state 0). The automaton has 10 transitions.

Figure 4: Failure factor oracle and failure factor storacle for $p = abcaab$.

As with the factor oracle and its failure version, the properties of the factor storacle other than acyclicity can easily be shown to hold for the failure factor storacle.

4 Empirical results

We implemented the four construction algorithms in Java, and ran benchmarks on an 1.7 GHz Intel Core i5 with 4 GB of 1333 MHz DDR3 RAM, running OS X 10.8.3. Two sets of data were used for the benchmarks, one consisting of generated strings of certain lengths, and one consisting of English words of widely varying lengths.

The first set consists 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.)

Figure 5 shows the distributions of the number of transitions for the first data set, for $m = 4..9$. As can be seen from the figure, the number of automata for particular numbers of transitions may vary drastically, from near 0 to almost m^m. Note that absent bars indicate values of 0 (i.e. no automata/keyword result in automata of a particular type with the given number of transitions), while bars represented by a flat line (i.e. bars seemingly of height 0) in fact indicate small but non-0 numbers of keywords/automata having the given number of transitions. As keyword length grows, it becomes easier to see that storacle versions of the automata are typically outperformed by oracle versions of the automata in terms of number of transitions. The graphs also show the *average* number of transitions per automata type for each of $m = 4..9$ (using dashed vertical lines). The average number of transitions for factor oracle and factor storacle on the one hand, and for their failure versions on the other hand, are fairly close, particularly for small word lengths, causing the dotted lines to overlap in the figures. Closely looking at the graph for e.g. $m = 8$ or $m = 9$ however shows that in fact four average lines are represented in each graph. The averages for factor oracles on the one hand and failure factor oracles on the other hand show that the use of failure version may lead to savings increasing from 1.5% for $m = 4$ up to 6.4% for $m = 9$, and suggest such savings may (sublinearly) increase further for longer keywords.

It is noteworthy that (for keyword lengths $m = 5..9$) only the factor storacle breaks the upper bound of $2m-1$ transitions established for the factor oracle; neither failure version breaks this barrier—and the failure factor oracle cannot for any keyword length, as it has the same transition set upper bound as the factor oracle. However, it is likely that the failure factor storacle will break the $2m - 1$ barrier as keyword length increases.

To make the experimental results more insightful, Figure 6 shows histograms for the *difference* in the number of transitions between factor oracles on the one hand and factor storacles, failure factor oracles or failure factor storacles on the other hand, again for all words of lengths $m = 4..9$ over alphabets of size m. For ease of understanding and comparison, the scale used here is a logarithmic one, and labels are in terms of percentages of all (automata for) keywords of a given length.

A number of interesting observations can be made from Figure 6:

– Comparing factor oracles and factor storacles, it turns out that in most cases, they are the same size. In a reasonable number of cases, growing to ca. 13% for $m = 9$, does the factor oracle have one or more transitions less than the factor storacle. What is remarkable is that only in rare cases does the factor storacle beat the

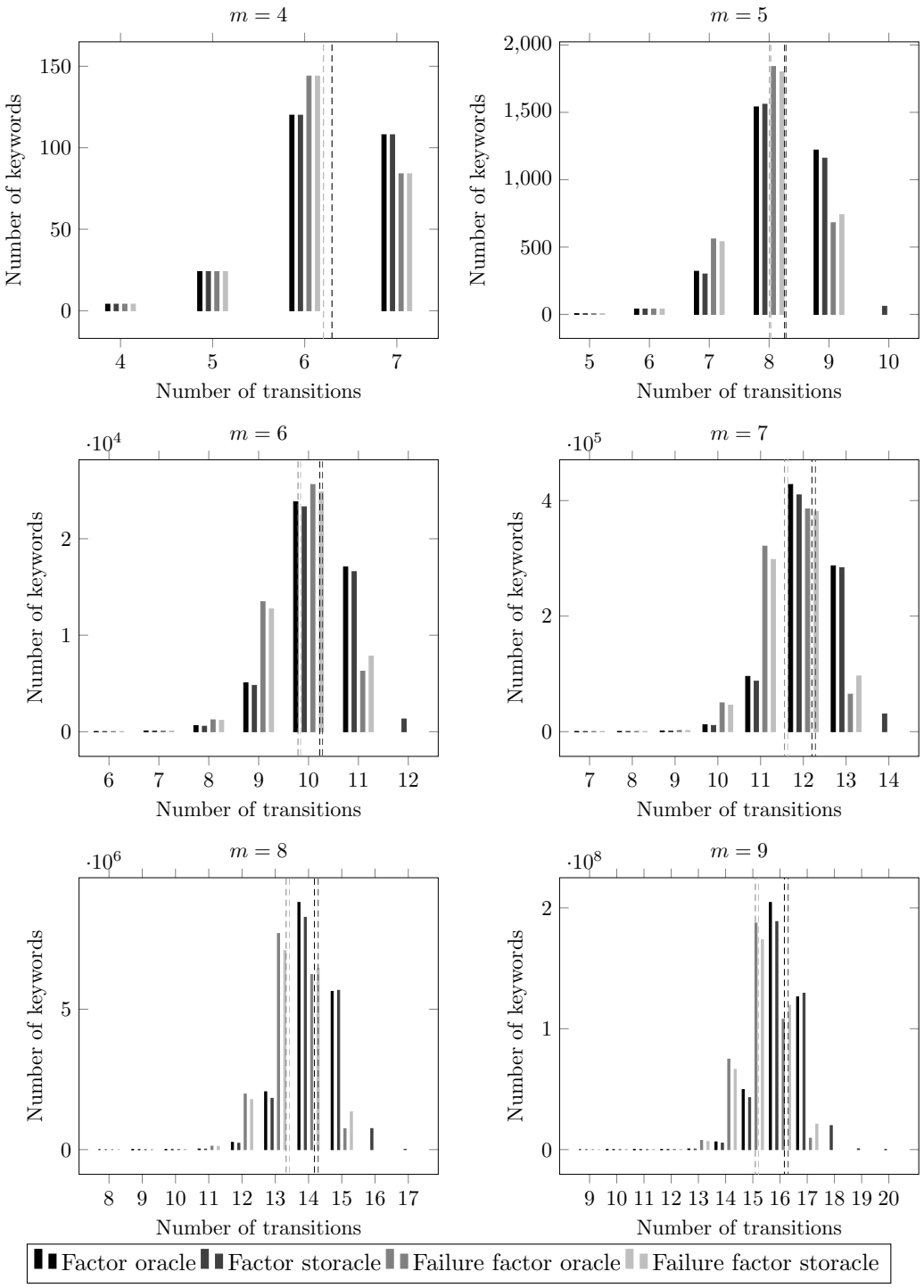


Figure 5: Distribution of number of transitions for the four automaton types, for all words of lengths $m = 4..9$ over alphabets of size m.

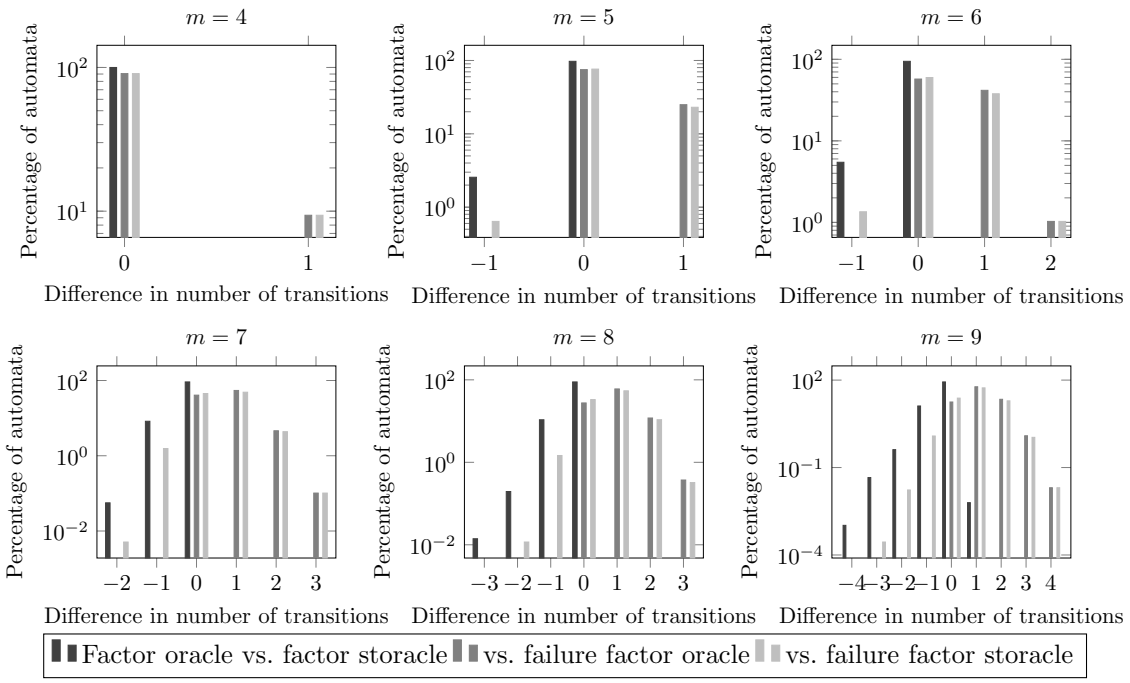


Figure 6: Distribution of difference in number of transitions for factor oracle versus each of the other three automaton types, for all words of lengths $m = 4..9$ over alphabets of size m.

factor oracle in size: no such cases occur for lengths $m = 4..8$, and it only happens in 0.006244% of cases for $m = 9$. As recollected in [6], the factor storacle was originally found by accident for $p = abcacdace$, for which it is *smaller* than the corresponding factor oracle. It thus turns out that this was somewhat of a lucky encounter as such cases are very rare. This also means that the use of a factor storacle is probably not advisable in general, compared to using a factor oracle.

– Comparing factor oracles and failure factor oracles, the experiments indicate that the failure version never performs worse than the original in terms of size, and frequently helps to reduce automaton size by 1 or 2 transitions for the keyword lengths considered. Larger savings seem to occur less frequently, although it is expected this will improve for longer keywords.
– Comparing failure factor oracles and failure factor storacles, the observation made for the non-failure cases above seem to hold true: it appears that failure factor oracles are preferable to failure factor storacles.

The second data set was obtained from [1]:

> "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 7 shows the distribution of the set (including words of length < 4).

Figure 8 shows the distribution of factor oracle and failure factor oracle sizes for the English words for the cases of words of lengths $m = 5, 7, 9, 11, 13$, and 15.

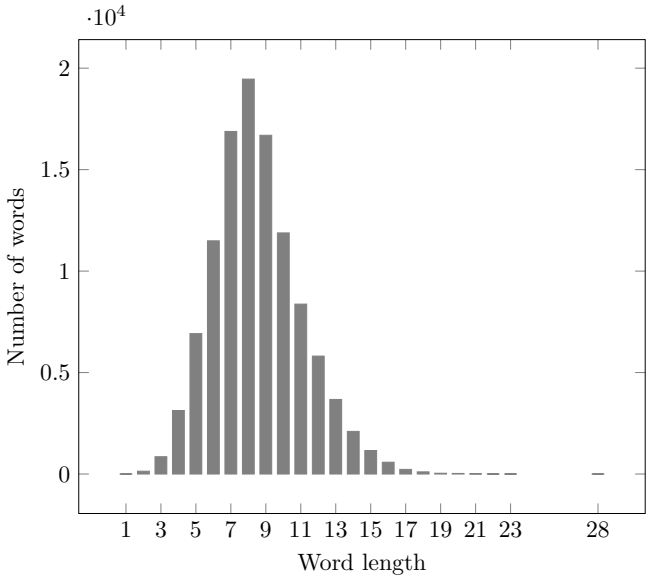


Figure 7: Distribution of lengths of the list of English words. No words of lengths 24−27 or of length over 28 occur. Words of length < 4 were not used for benchmarking.

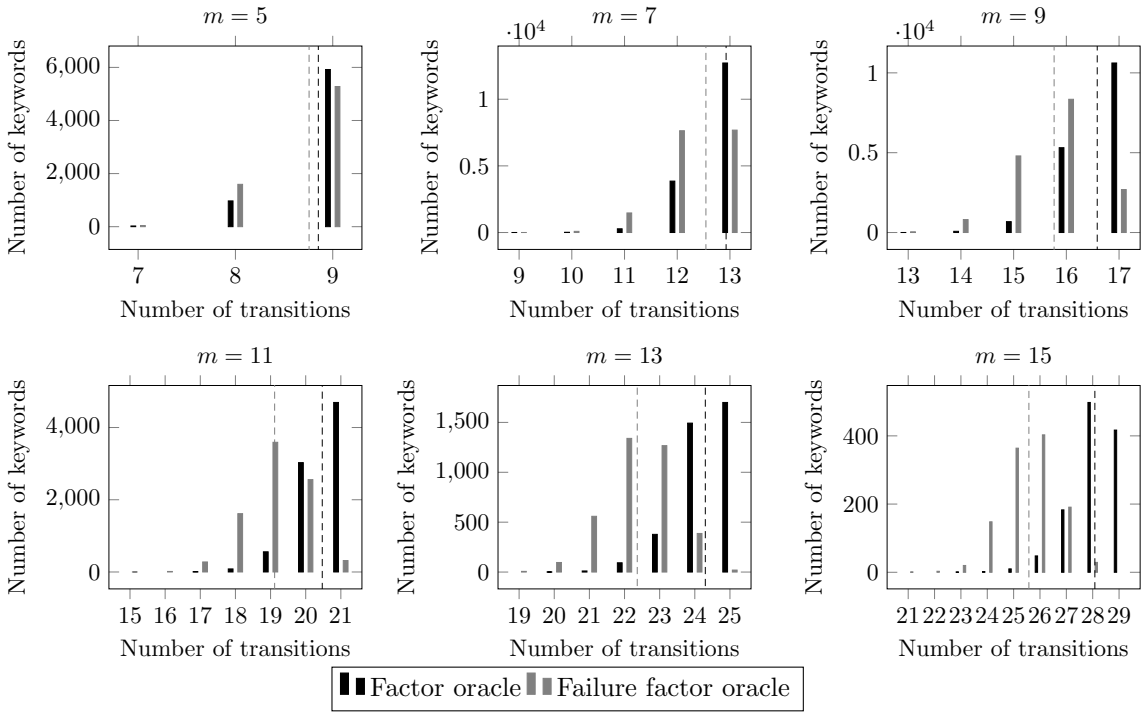


Figure 8: Distribution of number of transitions for the factor oracle and failure factor oracle automaton types, for English words of lengths $m = 5$ (6919 words), 7 (16882 words), 9 (16693 words), 11 (8374 words), 13 (3676 words), and 15 (1159 words).

Figure 9 shows the distribution of the difference in the number of transitions between the two automata kinds for the same data set. In that figure, the scale used is again a logarithmic one, and labels are in terms of percentages of all the (automata for) English words of a given length. (Results for the storacles versions are omitted, as

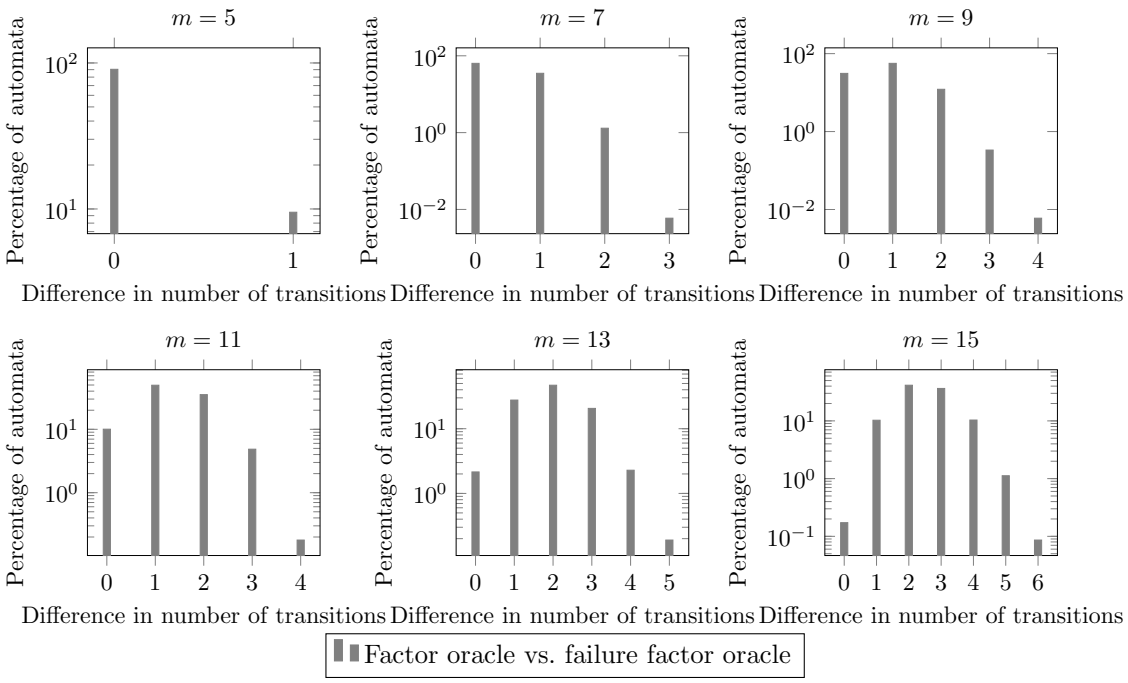


Figure 9: Distribution of difference in number of transitions for factor oracles versus failure factor oracles, for English words of lengths $m = 5$ (6919 words), 7 (16882 words), 9 (16693 words), 11 (8374 words), 13 (3676 words), and 15 (1159 words).

the discussion of results for the sets of generated strings indicated these automata are unlikely to be smaller than the corresponding oracle versions.) Comparing the results with those for the sets of generated strings, the following observations can be made:

- As was the case with the sets of generated strings, the use of failure transitions saves transitions compared to the non-failure automata versions. 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. Contrasting this with the results on the set of generated strings, the savings for a given word length are smaller, but as was the case there, the percentage of savings increases with increasing word length. This indicates that the use of failure transitions in weak factor automata may show particular promise in pattern matching for DNA processing and network intrusion detection, where longer patterns are typically being used.
- Compared to the results on the sets of generated strings, the results show fewer or no cases where the number of transitions is fairly close to the lower bound of m. This makes sense, as a language such as English has relatively few words with lots of repetition, while the generated sets contained many such strings, e.g. *aaaaaaa*, *abcabcabc*, *abccabccc* etc.
- As before for the sets of generated strings, the failure factor oracle for a given word never has more transitions, and often has fewer transitions than the corresponding factor oracle does. The distributions of differences also clearly show that with increasing word length, the distribution shifts further from a difference of 0, i.e. for longer words, the savings in number of transitions by the use of failure arcs increases.

5 Conclusions and Future Work

We have presented two new kinds of weak factor automata, based on modifications of two algorithms for constructing factor oracles and factor storacles. These new kinds of automata combine the use of failure transitions with the concept of the factor oracle and storacle, respectively.

Our experimental evaluation, with both generated strings of lengths up to 9 and English words of various lengths, showed that factor storacles and their failure versions are rarely competitive to factor oracles and their failure versions respectively. The results also show that with increasing word length, the savings in using failure factor oracles instead of factor oracles increase, to roughly between 5–9% for English words of lengths 9 and 15. Although not substantial, in restricted memory settings such savings may be useful. The increase in savings with increasing keyword length suggests more substantial savings may occur in the setting of DNA processing or intrusion detection, where patterns are typically longer than in natural languages.

A number of open questions w.r.t. the failure factor (st)oracles remain:

- What is the upper bound on the number of transitions for the failure factor oracle and failure factor storacle?
- How does the use of failure transitions in the failure automata constructions change the language accepted by the underlying non-failure automata constructions? Preferably the language accepted should not become much larger, as the use of weak factor automata in pattern matching applications becomes less efficient when more non-factors are accepted by such automata.
- While [3] introduced factor oracles for use in a particular pattern matching algorithm, it would be interesting to see how factor (st)oracles and failure factor (st)oracles can be used in a very different algorithm skeleton, such as the efficient *dead-zone* algorithm [16,18].
- The four types of oracle automata discussed here potentially accept *more* than just the factors of p, making them a type of *super* automaton. An alternative general technique for constructing super automata is discussed in [12,13,11]. How would a super automaton for the factors of p, constructed with those generalized techniques, perform against the (failure) factor (st)oracle of this paper?
- Language-preserving FDFA construction algorithms such as those in [14] and [4] could be applied to the constructed factor (st)oracles. Would the resulting failure factor (st)oracles differ significantly from those discussed in the present research?

References

1. *English wordlist*: `http://www.sil.org/linguistics/wordlists/english`.
2. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18 1975, pp. 333–340.
3. 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, 2001, pp. 51–72.
4. H. Björklund, J. Björklund, and N. Zechner: *Compact representation of finite automata with failure transitions*, Tech. Rep. UMINF 13.11, Umeå University, 2013.
5. J. Bourdon and I. Rusu: *Statistical properties of factor oracles.* Journal of Discrete Algorithms, 9(1) March 2011, pp. 57–66.
6. 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.

7. L. Cleophas, G. Zwaan, and B. W. Watson: *Constructing Factor Oracles*, in Proceedings of the Prague Stringology Conference 2003, Department of Computer Science and Engineering, Czech Technical University, Prague, September 2003.
8. L. Cleophas, G. Zwaan, and B. W. Watson: *Constructing Factor Oracles*, Tech. Rep. 04/01, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, January 2004.
9. L. Cleophas, G. Zwaan, and B. W. Watson: *Constructing Factor Oracles.* Journal of Automata, Languages and Combinatorics, 10(5/6) 2005, pp. 627–640.
10. L. G. W. A. Cleophas: *Towards SPARE Time: A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms*, Master's thesis, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, August 2003.
11. W. Coetser: *Finite state automaton construction through regular expression hashing*, Master's thesis, University of Pretoria, 2009.
12. W. Coetser, D. G. Kourie, and B. W. Watson: *On regular expression hashing to reduce FA size*, in Proceedings of the Prague Stringology Conference 2008, J. Holub and J. Žd'árek, eds., Czech Technical University in Prague, Czech Republic, 2008, pp. 227–241.
13. W. Coetser, D. G. Kourie, and B. W. Watson: *On regular expression hashing to reduce FA size.* International Journal of Foundations of Computer Science, 20(6) 2009, pp. 1069–1086.
14. 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.
15. A. Mancheron and C. Moan: *Combinatorial Characterization of the Language Recognized by Factor and Suffix Oracles*, in Proceedings of the Prague Stringology Conference 2004, Department of Computer Science and Engineering, Czech Technical University, Prague, August 2004.
16. M. Mauch, B. W. Watson, D. G. Kourie, and T. Strauss: *Performance assessment of dead-zone single keyword pattern matching*, in Proceedings of the Symposium of the South African Institute for Computer Scientists and Information Technologists, H. Gelderblom and H. Lotriet, eds., ACM International Conference Proceedings, Centurion, South Africa, Oct. 2012, pp. 59–68.
17. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Faculty of Computing Science, Technische Universiteit Eindhoven, September 1995.
18. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of dead-zone single keyword pattern matching*, in Proceedings of the 23rd International Workshop on Combinatorial Algorithms (IWOCA), S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science (LNCS), Heidelberg, Germany, 2012, Springer, pp. 236–248.