Noname manuscript No.
(will be inserted by the editor)

Computing the Shortest Reset Words of Synchronizing Automata

Andrzej Kisielewicz · Jakub Kowalski · Marek Szykuła

the date of receipt and acceptance should be inserted later

Abstract In this paper we give the details of our new algorithm for finding minimal reset words of finite synchronizing automata. The problem is known to be computationally hard, so our algorithm is exponential in the worst case, but it is faster than the algorithms used so far and it performs well on average. The main idea is to use a bidirectional BFS and radix (Patricia) tries to store and compare subsets. A good performance is due to a number of heuristics we apply and describe here in a suitable detail. We give both theoretical and practical arguments showing that the effective branching factor is considerably reduced. As a practical test we perform an experimental study of the length of the shortest reset word for random automata with up to $n = 350$ states and up to $k = 10$ input letters. In particular, we obtain a new estimation of the expected length of the shortest reset word $\approx 2.5\sqrt{n-5}$ for binary automata and show that the error of this estimate is sufficiently small. Experiments for automata with more than two input letters show certain trends with the same general pattern.

Keywords Synchronizing automaton, synchronizing word, Černý conjecture

1 Introduction

We deal with (complete deterministic) finite automata $A = \langle Q, \Sigma, \delta \rangle$ with the state set Q, the input alphabet Σ, and the transition function $\delta : Q \times \Sigma \to Q$. The action of Σ on Q given by δ is denoted simply by concatenation: $\delta(q, a) = qa$. This action extends naturally to the action qw of words for any $w \in \Sigma^*$. If $|Qw| = 1$, that is, the image of Q by w consists of a single state, then w is called a *reset* (or *synchronizing*) word for A, and A itself is called *synchronizing*. In other words, w resets (synchronizes) A in the sense that, under the action of w, all the states are sent into the same state. The synchronizing property is very important, because it makes the automaton resistant to errors that could occur in an input word. After detecting an error a synchronizing word can be used to reset the automaton to its initial state. Synchronizing automata have many practical applications. They are used in model-based testing (Broy et al, 2005), robotics (for designing so-called part orienters) (Ananichev and Volkov, 2003), bioinformatics (the reset problem) (Benenson et al, 2003), network theory (Kari, 2002), theory

This work was supported in part by Polish MNiSW grant IP 2012 052272. Main results of this paper were presented during the conference COCOON 2013 held in Hangzhou, China.

Department of Mathematics and Computer Science, University of Wrocław, Poland
E-mail: andrzej.kisielewicz@math.uni.wroc.pl, {kot,msz}@ii.uni.wroc.pl

of codes (Jürgensen, H., 2008) etc. The concept of synchronization appears also in other settings, such as synchronized data flow machines (Xue et al, 2000).

Theoretical research in the area is motivated mainly by the Černý conjecture stating that every synchronizing automaton A with n states has a reset word of length $\leq (n-1)^2$. This conjecture was formulated by Černý in 1964 (Černý, 1964), and is considered the most longstanding open problem in the combinatorial theory of finite automata. So far, the conjecture has been proved only for some special classes of automata and a general cubic upper bound $(n^3-n)/6$ has been established (see Volkov (2008) for an excellent survey of the results). Using computers the conjecture has been verified for small automata with 2 letters and $n \leq 11$ states (Kisielewicz and Szykuła, 2013) (and with $k \leq 4$ letters and $n \leq 7$ states (Trahtman, 2006); see also (Ananichev et al, 2010, 2012) for $n = 9$ states). It is known that, in general, the problem is computationally hard, since it involves an NP-hard decision problem. Recently, it has been shown that the problem of finding the length of the shortest reset word (*the reset length*, in short) is $\mathrm{FP}^{\mathrm{NP}[\log]}$-complete, and the related decision problem is both NP- and coNP-hard (Olschewski and Ummels, 2010) (cf. also (Berlinkov, 2010) and (Martyugin, 2009, 2011)). On the other hand, there are several theoretical and experimental results showing that most automata are synchronizing (Berlinkov, 2013) and most of them have relatively short reset words (Ananichev et al, 2010; Skvortsov and Tipikin, 2011).

In computing reset words, either exponential algorithms finding the shortest reset words (Kudłacik et al, 2012; Rho and F., 1993; Sandberg, 2005; Skvortsov and Tipikin, 2011; Trahtman, 2006) or polynomial heuristics finding relatively the shortest reset words (Gerbush and Heeringa, 2011; Kudłacik et al, 2012; Podolak et al, 2012; Roman, 2009b,a; Trahtman, 2006) are widely used. The standard approach is to construct the power automaton and to compute the shortest path from the whole set state to a singleton (Sandberg, 2005; Trahtman, 2006; Kudłacik et al, 2012; Volkov, 2008). Most naturally, the breadth-first-search method is used which starts from the set of all states of the given automaton and forms images applying letter transformations until a singleton is reached. Based on these ideas computation packages have been created (TESTAS (Trahtman, 2003) and recently developed COMPAS (Chmiel and Roman, 2011)). In (Roman, 2009a), Roman uses a genetic algorithm to find a reset word of randomly generated automata and thus obtains upper bounds on the reset length.

A new interesting approach for finding the exact length using a SAT-solver has been applied recently by Skvortsov and Tipikin (Skvortsov and Tipikin, 2011). The problem of determining if an automaton has a reset word of length at most l is reduced to the SAT problem and the binary search for the exact length is performed. Using this approach, the following experimental study is done. For chosen numbers n of states from the interval $[1, 100]$ hundreds of random automata with 2 input letters are generated, checked if they are synchronizing, and if so, the shortest reset word is computed. The results directly contradict the conjecture made by Roman (Roman, 2009a) that the mean reset length for a random n-state synchronizing automaton is linear and almost equal to $0.486n$. Skvortsov and Tipikin argue that their experiment based on a larger set of data shows that this length is actually sublinear and $\approx 1.95n^{0.55}$.

In this paper we present a new algorithm based on a bidirectional breadth-first-search. Implementing this idea requires efficiently solving the problem of storing and comparing resulted subsets of states. To this aim radix tries (also known as Patricia tries (Morrison, 1968)) are used. Also, a number of heuristics are applied that speed up the algorithm considerably. We analyze the algorithm from both theoretical and practical sides. As the first test of efficiency we have performed experiments analogous to those done by Skvortsov and Tipikin. Due to the well performance of the algorithm we were able to generate and check millions of binary automata up to 350 states, compared with $200-2000$ in (Skvortsov and Tipikin, 2011).

Our results confirm the hypothesis that the expected reset length is sublinear, but show that more precise is a smaller approximation $\approx 2.5\sqrt{n-5}$. In addition, the larger set of data enables us to estimate the error and to show that for our approximation with high probability the error is very small. We also verify and discuss other results and claims of (Skvortsov and Tipikin, 2011).

Our algorithm makes also possible to find a reset word of the shortest length (not only the reset length). Curiously, it works in polynomial time for known slowly synchronizing automata series (Ananichev et al, 2010). So far, most of the empirical research in the area concerns automata with 2 input letters. Our algorithm made possible to perform also experiments on automata with larger alphabets. These are of special interest because there is very little experimental material concerning synchronization of such automata. The results presented in the last section show certain trends with the same general pattern. It seems that, in spite of suggestions by some researchers, the behavior of automata does not change as the size of the alphabet increases.

The main results of this paper were announced in (Kisielewicz et al, 2013).

2 Main part of the algorithm

The algorithm gets an automaton $A = \langle Q, \Sigma, \delta \rangle$ with n states and k input letters. First, A is checked if it is synchronizing using the well known (and efficient) algorithm (Eppstein, 1990). If so, then we proceed to search for a synchronizing word of the shortest length. Here, one may perform the breadth-first search (BFS) on the power automaton of A starting from the set Q of all the states and computing successive images by the letters of the alphabet Σ (and recording the sequences of the letters applied). One may also search in the inverse (backward) direction starting from the singleton sets and computing successive preimages (this search will be referred to as IBFS). Both the searches have branching factor k (the number of input letters) and need to compute $O(k^l)$ sets (or $O(nk^l)$ in IBFS) to find a synchronizing word of the shortest length l. The idea behind bidirectional search is to perform two searches simultaneously and check if they meet. Then a synchronizing word may be found computing only $O(nk^{l/2})$ sets. However, to implement this idea there must be an efficient way to check each new subset to see if it already appears in the search tree of the other half of the search.

2.1 Maintaining lists of subsets

For each search we maintain the current list of subsets that can be obtained from the start in a given number of steps. Since the lists have a tendency to grow exponentially and to contain subsets obtained on earlier steps, it is more efficient to maintain additional lists of visited subsets (for each search) and to use them to remove from the current lists redundant subsets. We have checked experimentally that it is a good strategy to decrease the branching factor.

To check if the two searches meet one needs to perform *subset checking*: after each step, BFS or IBFS, we check if a set on the current IBFS list *contains* a set on the current BFS list. If so, it means that there are words $u, w \in \Sigma^*$ such that the image Qu is a subset of the preimage $\{q\}w^{-1}$ for some $q \in Q$. Consequently, $Quw = \{q\}$, as required.

Since, in the bidirectional approach, subset checking must be performed anyway, it may be also applied to reduce lists using the following simple observation. If S and T are subsets of Q such that $S \subseteq T$, then $|Tw| = 1$ implies $|Sw| = 1$ for any $w \in \Sigma^*$. It follows that, for example, a subset on the IBFS list contains a subset on the BFS list if and only if – with respect to inclusion – a maximal element on the IBFS list contains a minimal element on the BFS list. Consequently, the only subsets on the BFS lists we need to consider are those minimal with respect to inclusion and the only subsets on the IBFS lists we need to consider are those maximal with respect to inclusion.

To store and check subsets on the lists we apply an efficient data structure known as *radix trie* (Patricia trie) (Morrison, 1968). We show that the *subset checking* operation (checking whether a given set S has a subset stored in the trie) and the dual *superset checking* (checking whether a given set S has a superset stored in the trie) are efficient enough for these structures to make a combination of the ideas presented above work well in practice.

This approach is fast but memory consuming. In order to also make the algorithm work efficiently for larger automata, when the memory limit is reached, the bidirectional approach is replaced by a sort of an inverse DFS search not involving the tries of visited subsets anymore. We also apply several technical optimizations and heuristics which yields a considerable speed-up. They are described in Section 4.

2.2 Radix tries

A *radix trie* is a binary tree of the maximal depth n which stores subsets of a given n-set Q in its leaves. Having a fixed linear order of elements $q_1, \ldots, q_n \in Q$, each subset S of Q encodes a path from the root to a leaf in the natural way: after i steps the path goes to the right child whenever $q_i \in S$, and goes to the left, otherwise. A radix trie is *compressed* in the sense that instead in a leaf it stores a subset in the first node that determines uniquely the subset in the stored collection (no other subset shares the same path as a prefix of the encoding); c.f. (Morrison, 1968).

The insert operation for radix tries is natural and can be performed in at most n steps. The *subset checking* operation is performed by a depth-first-search checking if the given set $S \subseteq Q$ contains a subset stored in the visited leaf. An essential advantage is that the search does not need to branch into the right child of a node if the checked subset S does not contain the state corresponding to the current level. The superset checking operation (for IBFS) is done in the dual way. These issues are discussed in more detail in Section 6.

2.3 Basic procedures

A pseudocode of the algorithm is given in listings Algorithms 1–3. To make it clearer we restrict the task to finding the reset length only, i.e. the minimal length of a reset word. Yet, the algorithm can be easily modified to return also a reset word of such length (see 2.5).

We use, in principle, four radix tries T_c, T_v, T_{ic}, T_{iv} to maintain the BFS current, BFS visited, IBFS current, and IBFS visited lists, respectively. After initializing the tries we enter a loop consisting of at most `maxlen` steps (line 11). In each step we perform a step of the BFS procedure or IBFS procedure depending on comparison of estimated expected execution time of both steps, which we discuss in 4.2.

With no regard if BFS or IBFS step was performed recently, in lines 17-21 of Algorithm 1, the same goal test loop is performed. For each S in T_{ic}, the procedure T_c.CONTAINS_SUBSET_OF(S) is executed, which checks if T_c contains a subset of S. If so, we claim that l is the reset length for A. To prove this we need to analyze the content of the BFS and IBFS steps.

In BFS step (Algorithm 2), for each set S' in the current BFS trie and for each input letter a we compute the image $S = S'a$ and insert it to the list L. For each set $S \in L$ we check if a subset of S is already in the BFS visited trie. If so, we skip it. If not, we insert S into the BFS visited trie and into the (newly formed in line 9) BFS current trie T_c. Processing elements of L (line 10) in *ascending cardinality order* is a heuristic aimed in getting more subsets skipped in

Algorithm 1 The main part

```
Input  $A = \langle Q, \Sigma, \delta \rangle$ – a synchronizing automaton with $n = |Q|$ states and $k = |\Sigma|$ input letters.
Input  maxlen – maximum length of words to be checked.
                                          ▷ Initialize four radix tries to store and handle subsets of $Q$:
 1: $T_c \leftarrow$ EmptyTrie                                          ▷ BFS current trie
 2: $T_v \leftarrow$ EmptyTrie                                          ▷ BFS visited trie
 3: $T_{ic} \leftarrow$ EmptyTrie                                       ▷ IBFS current trie
 4: $T_{iv} \leftarrow$ EmptyTrie                                       ▷ IBFS visited trie
 5: $T_c$.insert($Q$)
 6: $T_v$.insert($Q$)
 7: for all $q \in Q$ do
 8:     $T_{ic}$.insert($\{q\}$)
 9:     $T_{iv}$.insert($\{q\}$)
10: end for
11: for $\ell \leftarrow 1$ to maxlen do
12:     if estimated time of the BFS step is smaller than that of IBFS then
13:         BFS_Step($T_c$,$T_v$)                     ▷ Modify BFS tries; minimize $T_c$ using $T_v$
14:     else
15:         IBFS_Step($T_{ic}$,$T_{iv}$)              ▷ Modify IBFS tries; minimize $T_{ic}$ using $T_{iv}$
16:     end if
17:     for all $S \in T_{ic}$ do                                      ▷ The goal test loop
18:         if $T_c$.contains_subset_of($S$) then
19:             return $\ell$                                          ▷ The reset length
20:         end if
21:     end for
22: end for
23: return "No synchronizing word of length $\leq$ maxlen"
```

Algorithm 2 BFS step procedure

```
 1: procedure BFS_Step($T_c$,$T_v$)
 2:     $L \leftarrow$ EmptyList                                  ▷ The list of all new images
 3:     for all $S' \in T_c$ do
 4:         for all $a \in \Sigma$ do
 5:             $S \leftarrow \delta(S', a)$               ▷ Compute the image of $S'$ by the letter $a$
 6:             $L$.insert($S$)
 7:         end for
 8:     end for
 9:     $T_c \leftarrow$ EmptyTrie
10:     for all $S \in L$ in ascending cardinality order do
11:         if not $T_v$.contains_subset_of($S$) then
12:             $T_v$.insert($S$)
13:             $T_c$.insert($S$)
14:         end if
15:     end for
16:     if $T_v$ has grown large since the last reduction then
17:         $T_v$.reduce
18:     end if
19: end procedure
```

the checking subset procedure in line 11, and in consequence, to deal with smaller structures. It also guarantees that T_c contains only only minimal sets in terms of inclusion.

Lemma 1 *After each step of the main **for** loop of Algorithm 1 the trie T_c contains only minimal elements of T_v (not necessarily all of them) and similarly the trie T_{ic} contains only maximal elements of T_{iv}.*

Proof Consider the structures T_v, T_c in Algorithm 2 as families of subsets. First of all note that in each step we have $T_c \subseteq T_v$. Therefore, it is enough to show that there is no pair of different subsets T and S, such that $T \in T_v, S \in T_c, T \subset S$.

Let S be a subset inserted into T_v and T_c and assume for a contrary that there is $T \in T_v$ such that $T \subset S$. It is impossible that S was inserted after T because of the subset checking test in line 11. However if it is inserted before T then T must be inserted after S in the same BFS step. But since we consider sets in ascending cardinality order it follows that $|S| \leq |T|$, which is a contradiction with $T \subset S$.

The IBFS step in Algorithm 3 is analogous and so is the proof for the trie T_{ic}. We note only that checking for a superset of a given set S in a given tree (line 11) is dual indeed: the search does not need to branch into the left child of a node if the checked subset S contains the state corresponding to the current level.

After executing lines 10-15 of Algorithm 2 the trie T_v may contain some redundant subsets (which are not minimal with respect to inclusion). Therefore in lines 16-18 we have an additional procedure to reduce T_v completely.

The procedure T_v.REDUCE consists of two steps. First, we form a list of elements of T_v using a DFS-search from the left to the right (smaller subsets first). This guarantees that if S precedes T on the list then S does not contain T. Hence the only pairs of comparable elements on the list are those with S preceding T and $S \subset T$. In the second step we insert the elements from the list into the empty T_v depending on the result of subset checking performed before each insertion. This guarantees that if a subset S of T is inserted then T will be skipped on the later step. Hence the resulting trie T_v contains no comparable subsets, as required.

Unfortunately, this procedure applied for such a large trie as T_v (which may be of exponential size in terms of n) may be time-consuming. We found experimentally that if the trie has not grown too large since the last reduction it is more effective to process a larger trie rather than to perform the reduction. In our implementation we perform it after the first step and then only when T_v contains at least k^2 times more sets than it had after the last reduction (which corresponds to two steps of the worst case computation with branching factor $k = |\Sigma|$).

Algorithm 3 IBFS step procedure

```
1: procedure IBFS_STEP($T_{ic}$,$T_{iv}$)
2:     $L \leftarrow$ EMPTYLIST                              ▷ The list of all new images
3:     for all $S' \in T_{ic}$ do
4:         for all $a \in \Sigma$ do
5:             $S \leftarrow \delta^{-1}(S', a)$             ▷ Compute the preimage of $S$ by the letter $a$
6:             $L$.INSERT($S$)
7:         end for
8:     end for
9:     $T_{ic} \leftarrow$ EMPTYTRIE
10:    for all $S \in L$ in descending cardinality order do
11:        if not $T_{iv}$.CONTAINS_SUPERSET_OF($S$) then
12:            $T_{iv}$.INSERT($S$)
13:            $T_{ic}$.INSERT($S$)
14:        end if
15:    end for
16:    if $T_{iv}$ has grown large since the last reduction then
17:        $T_{iv}$.REDUCE
18:    end if
19: end procedure
```

The IBFS step in Algorithm 3 is dual and completely analogous. In line 10 ascending cardinality order is replaced by descending one, in line 5 we compute preimages instead of images, and in line 11 subset checking is replaced by superset checking.

2.4 Correctness

In order to prove the correctness of Algorithm 1, we introduce additional notation. Let T_c^i denote T_c after performing i steps of BFS, and let T_{ic}^j denote T_{ic} after performing j steps of IBFS. Similarly, let T_v^i denote T_v after performing i steps of BFS, and let T_{vc}^j denote T_{iv} after performing j steps of IBFS. We have the following

Lemma 2 *For each set $S \in T_c^i$ there is a word u of length i, such that $Qu = S$. Similarly for each set $T \in T_{ic}^j$ there is a word v of length j, such that $\{q\}v^{-1} = T$ for some state $q \in Q$.*

Proof The proof is by induction. For $i = 0$ the claim is true with the empty word. For $i > 0$, we note that all new sets S inserted into T_c^i are obtained by applying a letter a to a set $S' \in T_c^{i-1}$ (line 5 of Algorithm 2). By induction hypothesis, there is a word u' of the length $i-1$ such that $Qu = S'$. Hence, $u'a$ has length i and we have $Qu'a = S'a = S$, as required. The proof of the second statement is dual.

We can prove now

Theorem 1 *Given a synchronizing n-state automaton $A = \langle Q, \Sigma, \delta \rangle$, Algorithm 1 returns the reset length of A or reports that no reset word of length $\leq$ `maxlen` exists.*

Proof Let l be the length of the shortest reset words for A. First we show that the algorithm in order to report the length of a reset word in line 19 needs to perform at least l (BFS or IBFS) steps.

Assume that the algorithm reaches line 19 after i steps of BFS and j steps of IBFS. So there are sets $S \in T_c^i$ and $T \in T_{ic}^j$ such that $S \subseteq T$. By Lemma 2, there are words u, v of lengths i, j, respectively, and a state $q \in Q$ such that $Qu = S$ and $\{q\}v^{-1} = T$. Thus, $Quv = \{q\}$, and uv is a reset word of length $i + j$. Consequently, $l \leq i + j$.

Now we show that, if $l \leq$ `maxlen`, then the algorithm reaches line 19 after at most l steps. By induction, we prove the following more general statement implying our claim: *for each $i, j \geq 0$, $0 \leq i + j \leq l$, after i steps of BFS and j steps of IBFS there are sets $S \in T_c^i$ and $T \in T_{ic}^j$, and there exists a reset word $w = uxv$ of length l, where $|u| = i, |v| = j, |x| = l - i - j$, such that $Qu = S$ and $\{q\}v^{-1} = T$.*

For $i + j = 0$, because of the initialization in lines 5-10, we have that $Q \in T_c^0$ and $\{q\} \in T_{ic}^0$, and a reset word of length l is as required. Assume that the statement is true for all $i' + j' < i + j$. Assume also, first, that the $(i + j)$-th performed step is BFS one. Then, by the induction assumption there exists a reset word $w' = u'x'v$ of length l and sets $S' \in T_c^{i-1}$ and $T \in T_{ic}^j$ such that $Qu' = S'$ and $\{q\}v^{-1} = T$ for some state $q \in Q$, $|u'| = i - 1, |v| = j$.

Since $i + j \leq l$, $|x'| > 0$. Let a be the first letter of x' and $x' = ax''$. We need to consider two cases, depending on whether $S'a = \delta(S', a)$ (created in line 5 of Algorithm 2) is added (in line 13) into T_c^i or not. If so, then the statement is true, because we have the reset word $w = w' = (ua)x''v$ and sets $S = S'a \in T_c^i$ and $T \in T_{ic}^j$, with required properties..

Otherwise the reason for not adding $S'a$ into T_c^i must be a set $S \in T_v^i$, such that $S \subseteq S'a$ (line 11). Let u be the word corresponding to S by Lemma 2. Then the word $w = ux''v$ (where $x' = ax''$) is a reset word. If $|u| < i$ (we do not know yet if $u \in T_c^i$), then w is shorter than l, because $|u| + |x''| + |v| < i + (l - (i - 1) - j - 1) + j = l$, which is a contradiction. So, $|u| = i$, which means that S has been added into T_v^i in the currently performed i-th BFS step. It follows that S has been also added into T_c^i. Now, $w = ux''v$ is the required word for i, j with $S \in T_c^i$ and $T \in T_{ic}^j$, $Qu = S$, and $\{q\}v^{-1} = T$.

For the second part of the proof we need to assume that the $(i + j)$-th performed step is IBFS one. In this case the proof is, again, completely analogous. The difference is that by the induction assumption, we have now a reset word $w' = ux'v'$, and we take into consideration the last letter of x'. We leave this part to the reader.

2.5 Finding a shortest reset word

In order to find also a reset word of the minimal length l, one needs to apply the following slight modification to the algorithm described above. The main point is that together with the sets stored in the current tries T_c and T_{ic} we need to store also the words assigned to these sets by Lemma 2. To this end, in line 5 of Algorithms 2–3 we assign to S the word obtained by concatenating the word assigned earlier to S' with the letter a (at the end or at the beginning, respectively). When the goal is reached, the two words are simply merged to form the required reset word. Of course, instead of complete words, with each set we can store only a letter and a pointer to the previous part of the word. From these the word is reconstructed when we reach the goal. We note that in this way the asymptotic time and space complexity of the algorithm remain the same.

3 The full algorithm

In the full version of the algorithm we first check whether the input automaton is synchronizing at all, and if so, we try to find any reset word using fast heuristic algorithms in order to obtain a starting value for `maxlen` in Algorithm 1 bounding from above the reset length. In case when no reset word is found quickly by the heuristic algorithms, `maxlen` is set to $(n-1)^2 + 1$, so that the algorithm returns either the reset length or a counterexample to the Černý conjecture.

The bidirectional BFS search works if we have no limit on memory resources. Since the number of sets stored in the tries grows exponentially with the number of steps performed, for large automata, we can easily run out of memory. To deal with this, we change the search strategy when we reach the memory limit. Rather than to continue BFS searches in the both directions we switch to depth-first search, which has restricted memory requirements, and may use the subsets and words computed so far. This final phase of the algorithm may be used also to reduce the computation time of the algorithm (even if we are far from reaching the memory limit). This will be discussed in subsection 4.7.

Our experiments show that it is more efficient to apply the *inverse* DFS (IDFS), that is, one starting from the sets in T_{ic} and computing the preimages to find a set containing a member of T_c (rather than one starting from the sets in T_c and computing images to find a set contained in a member of T_{ic}). An important modification is that we perform search on partial lists of subsets making use of all available memory rather than on single subsets. This gives an additional boost.

3.1 The IDFS phase

Algorithm 4 shows a more formal description of this final phase. It is a recursive procedure working on the list L_{ic} of current sets (which in the first step is obtained from the trie T_{ic}) and modifying the variables `depth` and `minlen`. The first represents the current depth of the search (including all the steps of BFS and IBFS performed during the bidirectional search phase). The second represents the length of the reset word found so far. It is used to bound the depth of the IDFS search. We do not need to perform `depth` $\geq$ `minlen` steps, since we have found already a reset word of length `minlen`. In line 7, `minlen` is decreased after each case when a shorter reset word is found, and at the end of the procedure contains the length of the shortest reset word. The procedure uses the T_c trie for the subset checking (line 6). The memory for storing the other tries is released, and the trie T_{ic} is replaced by the list L_{ic} of subsets (cf. the general description in Algorithm 5, lines 14-15).

The procedure IDFS starts computation with `depth` equal to the number of steps performed in the bidirectional BFS search increased by one. It gets a list L_{ic} of sets and computes the next list, containing all preimages which can be obtained from sets in the input list (lines 3-5). The next list L'_{ic} is therefore k times larger than the input list and it is split up into several parts, so that each of the partial lists does not exceed the maximum allowed size `maxsize` (line 18). Then the search branches recursively for each of the smaller lists. The trie T_c is not changed during the process, and is used for the goal test performed for each new generated set, while inserting it into the next list (line 6). In case the goal test is positive it means we have found a new shorter reset word. Then `minlen` is modified suitably and new (greater) `maxsize` is computed taking into account the new value of `minlen` and available memory.

In lines 3 and 17 the list is sorted in descending cardinality order, which is a heuristic method to reach the goal faster and so to reduce the depth of the search. Note that the sorting here can be performed in linear time by counting sort.

Algorithm 4 The IDFS recursive procedure

```
Input $L_{ic}$ – current list of sets
      depth – current (total) depth of the search
      minlen – length of the found reset word; used to bound the search depth
      maxsize – the number of sets allowed in partial list of $L_{ic}$
 1: procedure IDFS($L_{ic}$,depth,minlen, maxsize)
 2:     $L'_{ic} \leftarrow$ EmptyList
 3:     for all $S \in L_{ic}$ in descending cardinality order do
 4:         for all $a \in \Sigma$ do
 5:             $S' \leftarrow \delta^{-1}(S, a)$                     ▷ Compute the preimage of $S$ by the letter $a$
 6:             if $T_c$.contains_subset_of($S'$) then                         ▷ The goal test:
 7:                 minlen $\leftarrow$ depth                                ▷ modify minlen suitably
 8:                 Compute new maxsize                        ▷ using minlen and available memory
 9:                 return minlen
10:             end if
11:             $L'_{ic}$.insert($S'$)
12:         end for
13:     end for
14:     if minlen $- 1 =$ depth then                          ▷ a new reset word found in lines 6-7
15:         return minlen
16:     end if
17:     Sort $L'_{ic}$ descending by cardinality.
18:     Split $L'_{ic}$ into a sequence of partial lists of maximal size maxsize.
19:     for all partial lists $L$ of split $L'_{ic}$ do
20:         minlen $\leftarrow$ IDFS($L$,depth $+ 1$,minlen,maxsize)       ▷ By each call minlen can be decreased
21:     end for
22: end procedure
```

It should be clear that if the bidirectional search Algorithm 1 performed t steps of BFS and IBFS and did not found a reset word, then Algorithm 4 started with `depth` $= t + 1$ and suitable values of the remaining variables completes the job correctly. As mentioned at the beginning of the section, `minlen` is set to the length of a reset word found by heuristics algorithms or to the Černý bound $(n-1)^2+1$, and `maxsize` is computed on the basis of available memory. Then, if the automaton A happens to have a reset word of the length $t+1$, procedure IDFS$(L_{ic}, t+1, \texttt{minlen}, \texttt{maxsize})$ passes the goal test in line 6, and terminates with the value $\texttt{minlen} = t+1$ equal to the length l. Otherwise, recursive calls in line 20 perform a complete (inverse) depth first search restricted by `minlen`. Decreasing `minlen` whenever a shorter reset word is found decreases the number of visited subsets.

The description of the full algorithm is given in Algorithm 5 (it includes also some heuristics that will be described later). If $\texttt{minlen} = t+1$, that is the equality $\ell = \texttt{minlen} - 1$ holds in line 13, then IDFS is not called at all, because we have found already a reset word of length `minlen` (or we know that no reset word of length $(n-1)^2$ exists). Otherwise, IDFS performs its recursive calls.

Algorithm 5 The algorithm

```
Input  $A = \langle Q, \Sigma, \delta \rangle$ – a $n$-state automaton on $k$ letters
                                                              ▷ Preprocessing
 1: if $A$ is not synchronizing then                          ▷ Use Eppstein algorithm
 2:     return "Not synchronizing automaton"
 3: end if
 4: minlen ← $(n-1)^2+1$                                      ▷ Restrict search to the Černý's bound
 5: minlen ← min(minlen, EppsteinAlgorithm($A$, minlen))
 6: minlen ← min(minlen, FastSynchro($A$, minlen))
 7: minlen ← min(minlen, CutoffIBFS($A$, minlen))
                                                              ▷ Bidirectional Search
 8: Reorder the states of $A$ using the Markov chain model
 9: $\ell$ ← BidirectionalSearch($A$, minlen $-$ 1)           ▷ Algorithm 1
10: if a reset word was found then
11:     return $\ell$                                         ▷ The reset length
12: end if
                                                              ▷ IDFS
13: if $\ell <$ minlen $-$ 1 then                             ▷ search for possible shorter reset words
14:     Free tries $T_v$ and $T_{iv}$
15:     Transform $T_{ic}$ into the list $L_{ic}$ and free $T_{ic}$
16:     Reorder the states of $A$ using $T_c$
17:     Compute maxsize – currently allowed size for IDFS partial lists
18:     minlen ← IDFS($L_{ic}$, $\ell+1$, minlen, maxsize)     ▷ Algorithm 4
19: end if
20: if minlen $\le (n-1)^2$ then
21:     return minlen                                         ▷ The reset length
22: else
23:     return "No synchronizing word of length $\le (n-1)^2$"   ▷ Reaching this line means that the Černý
    conjecture is false
24: end if
```

3.2 Using other heuristic algorithms

In our implementation we use Eppstein algorithm (Eppstein, 1990), FastSynchro algorithm (Kudłacik et al, 2012) and our own procedure Cut-Off IBFS (Kowalski and Szykuła, 2013). The latter is the standard IBFS search with cutting the branches of the search that do not seem promising (that do not increase the sizes of subsets fast enough). The order here is from the fastest algorithm to the slowest one, and from the worst to the best one in terms of finding a bound as small as possible. In this order, slower algorithms use a previously found bound by a faster algorithm to terminate computation when the bound is achieved. Our procedure Cut-Off IBFS often finds very good bounds, but works relatively slow when the input bound is large.

As given in Algorithm 5, Eppstein algorithm is used also at the beginning to check synchronizability. After that we assume the initial bound $(n-1)^2+1$, to be able to discover a counterexample for the Černý conjecture (if it is the case; see line 23). The minimal bound `minlen` found by heuristic algorithms decreased by one is used as `maxlen` for bidirectional search (line 9). If this procedure does not find a shorter reset word then it means that it has been found already by heuristic algorithms (and has length `minlen`) or that no reset word of length less than $(n-1)^2$ exists (the latter is in case when heuristic algorithms did not find such a word either; lines 20-23). This makes possible to spare the last step in bidirectional search and gives a boost if `minlen` is in fact the minimum length. More importantly, using heuristic algorithms to obtain a good initial bound is a part of the optimization described in 4.7.

3.3 Working with limited memory

Combining the bidirectional search with the IDFS phase guarantees that the algorithm will not exceed a certain memory limit, which is important in practice. The more memory is provided for the algorithm the more efficient computation it performs. When measuring memory usage we need to consider stored sets in the tries and lists, and nodes in the tries. The other structures used by the algorithm can be bounded by $O(kn^2)$, including the automaton itself and the memory used by heuristic algorithms in the preprocessing phase.

When during the bidirectional phase the tries and lists reach the memory limit then we switch to IDFS phase. We can then free the visited tries. In the IDFS phase we need to store the sets and nodes of T_c and sets from the lists L_{ic} at each level. There are at least $2k|T_{ic}|+d$ sets in the lists, where d is the difference between the upper bound and the number of currently performed steps. This comes from the fact that we decrease the size of a partial list in a IDFS step at most two times, and we also need at least one set for a recursive call of the IDFS step procedure. This can be bounded by $O(n^2)$, and so the minimum required memory for all recursive calls of the IDFS step procedure is $O(n^3)$. Remaining available memory is used to make L_{ic} lists larger.

The whole algorithm runs in $O(kn^2)+\min\{O(n^3),C\}$ space, where C is a parameter determining the memory limit. The larger the limit is the longer the bidirectional phase works, and the larger L_{ic} lists are in the IDFS phase. So the larger C the faster execution of the algorithm is.

4 Heuristics and optimizations

In this section we describe the most important heuristics and optimizations added to the generic version of the algorithm. For computationally hard problems, heuristic improvements are widely used and can yield in an impressive speed-up in an average case (see for example Batsyn et al (2013)). In our algorithm, altogether they can reduce computation time by as much as 96% and 99% for automata with $n=150$ states and $n=200$ states, respectively (that is by a factor of from 25 to 100 and more), relative to the implementation without these optimizations. In order to estimate roughly speed-up of a given optimization we compare the performance of the full version of the algorithm with and without this optimization. This is done mainly on the sample of a few hundreds of random automata with $n=150$ and $n=200$ states. We note that this must be considered only a rough estimation, since some of these heuristics may be argued to have better impact for larger automata. As most of computation time is taken by subset-checking, the majority of heuristics are aimed to optimize these operations.

4.1 Technical optimizations

We start from mentioning two obvious technical optimizations. First of all, since every synchronization leads to a state in the *sink component* of the automaton (that is, in the minimal strongly connected component of the underlying digraph), we may replace the set Q in the initialization of IBFS in line 7 of Algorithm 1 by the set Q' of the states in the sink component, that have at least 2 incoming edges on some letter. At second, we should mention that we store sets as *bit-vectors*, which minimizes the used memory and provides a constant time checking and inserting a state in a set. (See (Kudłacik et al, 2012) for a discussion of other possible encodings).

Another technical optimization is based on precomputing images. This idea was applied in (Kudłacik et al, 2012, 5.1.12). Before we run the algorithm, we split up the states of the automaton into groups of size at most t. Then for each group, each possible subset of states from the group and each letter, we compute the image and the preimage of the subset. Having these images, we can compute the image of a set by using only $\lceil n/t \rceil$ union operations instead of computing n transitions. However computing transitions can be done in constant time, while union depends highly on the maximum size of bit-vector which can be processed in an elementary operation; for example it can take $O(n/64)$ in 64-bit architecture. Nevertheless for our automata sizes setting $t = 8$ provides a speed-up by an average of 21% and 16% for automata with $n = 150$ and $n = 200$ states, respectively.

4.2 Estimation of expected step time

To decide which step will be performed in line 12 of the Algorithm 1 we follow the greedy strategy choosing this step whose execution time, together with the goal test, seems to be smaller at the moment. This strategy would be optimal if execution times of particular steps of one kind would be independent of those of the other kind. This is not the case because of the goal test is taking into account. Nevertheless, we have checked it experimentally, that this strategy leads to a significant improvement.

We use a rough estimation of expected execution time by calculating upper bounds for the expected number of visited nodes in subset checking operations, under some simplifying assumptions. Since all other operations in the steps in question are linear in terms of n and the sizes of the current lists, subset checking are the most time consuming operations. Therefore for estimation of the BFS step we take simply the sum $\textsc{ExpBfs} = E_c + E_v$ of the expected number of visits nodes in the tries T_v (inside the BFS step) and T_c (inside the goal test), respectively. Similarly, for the IBFS step we take the sum $\textsc{ExpIbfs} = E_{iv} + E_{ic}$ of the expected number of visited nodes in the T_{iv} and T_{ic}, respectively.

To estimate E_v, E_c, E_{iv}, and E_{ic} we apply the formula established in Theorem 4 in Subsection 6.1. We assume that on each step the subsets in the tries are random with the uniform Bernoulli distribution. Assuming that (on a given step) a set $S \subseteq Q$ contains each element of Q with independent probability p, and for every set S' in the trie in question the probability of containing any element is q, for m sets in the trie, we have that the expected number of visited nodes during the subset checking operation does not exceed

$$\textsc{ExpNvn}(m, p, q) = \left(\frac{1+p}{p} + \frac{1}{q - pq} \right) m^{\log_w (1+p)},$$

where $w = \frac{1+p}{1+pq-q}$.

Now, to compute the probabilities p and q we count the average size of the subsets in each of the tries and divide by $n = |Q|$ (the maximal number of the elements). For the BFS step we first perform the subset checking in the trie T_v, which grows during the step (lines 11-14 of Algorithm 2). The cardinality of T_v may increase as much as to $|T_v| + k|T_c|$. The sum of the cardinalities of the sets in T_v may increase as much as to $\sum_{x \in T_v} |x| + k \sum_{x \in T_c} |x|$. We found experimentally, that the most efficient is to take these upper bounds to represent the average probability p_v, for any element, to belong to a set in T_v during subset checking in lines 11-14:

$$p_v = \frac{1}{n(|T_v| + k|T_c|)} \left(\sum_{x \in T_v} |x| + k \sum_{x \in T_c} |x| \right).$$

For the goal test the probability p_c for an element to belong to a set in T_c, after modifying T_c in the BFS step, may be defined as

$$p_c = \frac{1}{nk|T_v|} \sum_{x \in T_v} k|x| = \frac{1}{n|T_v|} \sum_{x \in T_v} |x|.$$

Note that this is the same as before modifying T_c. So the same probability may be used in the goal test after the IBFS step. Analogously we define probabilities p_{ic} and p_{iv} for any element to belong to a set in T_{ic} and T_{iv}, respectively.

$$p_{ic} = \frac{1}{n|T_{ic}|} \sum_{x \in T_{ic}} |x|$$
$$p_{iv} = \frac{1}{n(|T_{iv}| + k|T_{ic}|)} \left(\sum_{x \in T_{iv}} |x| + k \sum_{x \in T_{ic}} |x| \right).$$

Using these we define

$$\textsc{ExpBfs}(T_c, T_v, T_{ic}) = k|T_c|\textsc{ExpNvn}(|T_v| + k|T_v|, p_c, p_v) + |T_{ic}|\textsc{ExpNvn}(k|T_c|, p_{ic}, p_c)$$
$$\textsc{ExpIbfs}(T_{ic}, T_{iv}, T_c) = k|T_{ic}|\textsc{ExpNvn}(|T_{iv}| + k|T_{iv}|, 1 - p_{ic}, 1 - p_{iv}) + k|T_{ic}|\textsc{ExpNvn}(|T_c|, p_{ic}, p_c)$$

Depending on which of these values is smaller we perform the BFS or IBFS step, respectively.

In our empirical observations this heuristic reduces computation time by an average of 9% and 26% for automata with $n = 150$ and $n = 200$ states, respectively, relative to the implementation performing the BFS and IBFS steps alternatingly (or when the choice of the step is based merely on the sizes of the current tries.) It usually leads to perform slightly more BFS steps, since average sizes of subsets decrease faster in BFS than increase in IBFS. By a result of Higgins after applying two BFS steps the average size of subsets is as small as $0.55n$ (see (Higgins, 1988)). Our empirical observations show that the two searches meet when the sizes of subsets are about 0.09 for automata with $n = 200$. This fact is also the reason why in the goal test we decided to use subset checking in T_c rather than superset checking in T_{ic} (subset checking does not require branching in subtries corresponding to elements not belonging to the queried set).

4.3 Reduction of the automaton

If the input automaton is not strongly connected, after some steps of BFS it can be reduced to a smaller automaton without the states not involved in computation anymore. More precisely, we can remove the states which are not reachable from any state in any subset in the current BFS list. Smaller automata lead obviously to faster execution because of having smaller tries and faster computation of images and preimages (when stored as bit vectors or other constant-space representations).

So, at the beginning, before the main loop of Algorithm 1 (line 11), we perform a few steps of BFS and when the size of T_c is larger than $s|Q|$, for an experimentally established constant s (we use $s = 16$), we check if there are unreachable states in Q (that is, the states which cannot be obtained by applying any word to any state in any set in T_c). This is done by the standard DFS search on Q. If this is the case, we create a reduced automaton A' removing the unreachable states, and rebuild all the tries to make them compatible with the reduced automaton. Then, the algorithm may continue using the parameters computed so far.

Our experiments show that after the first reduction the automaton is usually strongly connected (and no further reduction of this kind can be done). Yet, this optimization is efficient since we have proved that the fraction of strongly connected automata to all automata with n states tends to 0 as n goes to infinity, and that the size of the strongly connected component is on average close to $1 - 1/e^k$. From our experiments it follows that for synchronizing automata with $k = 2$ this size is $\approx 0.7987n$. Thus, for example, automata with $n = 200$ states are reduced on average by as much as 40 states, which results in a speed-up of 27%.

4.4 Reordering of the states

Efficiency of operations on radix tries depends on the order in which the input automaton's states are processed. Since all the tries change during the search it is difficult to find the optimal ordering of the states. Generally, it seems that the subset checking should be performed faster if the states occurring more frequently in queried subsets are later in the ordering. A heuristic argument is that radix tries have usually logarithmic height for a wide class of distributions (cf. (Devroye, 1982)), and therefore the states at the end in the ordering are rarely or never checked. As a result, the "effective size" of the queried sets is smaller (provided they are large enough). Also, if a state occurs rarely and with a high probability it is not a member of a given queried set, the search for a subset goes only into one branch on the level of this state (but on the other hand, in such a case, the branch is relatively small). Which of these arguments is prevailing may be decided experimentally.

We have tried to find frequencies of occurrences of states, and a preferred initial order based on them, by a heuristic approach using stationary distribution of a Markov chain created for the automaton (see (Stewart, 1994) for general use of Markov chains in computations). We first find the sink component of the automaton, which can be done quickly using the well-known Tarjan's algorithm (Tarjan, 1972). We define the probability of transition from a state a to a state b in the sink component as follows. For each letter which takes a to b, we define the probability to be $1/k$ plus ϵ, for some small $\epsilon > 0$. Then they are normalized to be summed up to 1. A Markov chain has a stationary distribution if it is irreducible and ergodic, that is, there exists a finite number N such that any state is reachable from any other state with positive probability after exactly N steps. Because we used the sink component, the Markov chain is irreducible and because we set non-zero probability (due to adding ϵ) for each possible transition the Markov chain is ergodic. The stationary distribution of the Markov chain is computed in a direct way (see (Stewart, 2000)).

Now, the set of states is reordered in such a way that the states are sorted ascending by probabilities in the computed stationary distribution. The states which do not belong to the sink component are placed at the end, sorted ascending by in-degree (they usually play little role in differentiating the subsets). Radix tries with subset checking use this very order of states and radix tries with superset checking use the inverse order. This optimization is performed once before the bidirectional search phase (Algorithm 5, line 8). Choosing this very ordering as a preferred one has been confirmed by experiments with various trials.

The situation changes during the IDFS phase, when the trie T_c is fixed and does not change anymore. The frequencies of occurrences of the subsets in T_c may by computed exactly. It seems to give better performance of subset checking when the states are ordered descending: from the most frequently occurring in the trie to those occurring the least. This has been confirmed strongly by experiments. In our tests, both reordering reduces computation time by an average of 16% and 26% for automata with $n = 150$ and $n = 200$ states, respectively. Because of the relatively high cost of computing the first reordering, it seems that this part may prove to be more profitable for automata with larger number of states.

4.5 Additional lexicographical ordering

In Algorithm 2 line 10 we sort the list in ascending cardinality order. This helps us reducing the size of T_v. In addition to that for sets with the same cardinalities we sort them by inverse lexicographical order, that is S is before T if and only if the first state (in the applied automaton's order) which differs them is in S and not in T.

The reason for this is the following. Consider the operation in line 10. Let $S, T \in L$ be sets with the same cardinalities, and let $q \in Q$ be the first state differing S and T with $q \in S$ and $q \notin T$. Assume that S precedes T in L. Now, if S is inserted in T_v, then during the subset checking for T, when it reaches the level of q in the trie, it does not go into the branch of S (and additional nodes created by inserting S), since no subset of T contains q. Also, if S is not inserted in T_v no additional nodes are visited during the subset checking for T. In the opposite case, when T precedes S, and T is inserted in T_v, subset checking for S must go additionally into the branch created for T.

The situation for superset checking is dual. This optimization reduces computation time only for very large automata, because of more expensive cost of sorting. For automata with $n = 250$ states it gives a speed-up by an average of only 3%, and for automata with $n = 300$ of 4%. However, it seems that it may have even bigger impact in case of larger automata.

4.6 Sub-tries elimination in subset-checkings

This heuristic is used to reduce the number of visited nodes during subset-checking, by skipping the nodes whose sub-tries certainly do not have a subset of the queried set. In each node of a trie we can store additionally the minimum size of all the sets stored in the sub-trie. When checking for a subset, if this size is larger than the subset's size, we can skip it and do not go down. In addition to storing a single size, we may store a marker for each state indicating if all the stored sets contains this state. If so, and if such a state does not occur in the checked subset, we can skip the node with such a marker.

Others combinations are possible. For each family of subsets of the states, we can store for each subset the minimum number of elements in the stored sets. However we use this only for the whole set Q and the singletons, since these cases can be efficiently implemented. They both provided a significant reduction of tries traversal, resulting in speed-up by an average of 49% of computation time for automata with $n = 150$ states, and 51% for automata with $n = 200$ states.

A disadvantage of this heuristic is that it uses a lot of additional memory, which may result in earlier switching to the IDFS phase. It requires $O(n)$ space for a node and $O(n)$ time for visiting a node during subset-checking, instead of $O(1)$ time and space in the version without the heuristic.

4.7 IDFS shortcut

As mentioned at the beginning of Section 3, the final IDFS search may be used to reduce the computation time by several orders of magnitude. To this end one needs to observe that knowing that bidirectional search is close to end it is profitable to switch to IDFS phase: at the end the IDFS works much faster, since we do not need to check visited sets and do not need to reconstruct T_c anymore. We call this optimization the *shortcut*. Between steps we use an estimate if it is faster to continue the bidirectional phase or to switch to IDFS phase. Note that the IDFS has a lower constant factor, but the branching factor is equal to k. So, it slows the search if started too early.

Let d be the number of steps remaining to finish the bidirectional phase search. We use formulas defined in Subsection 4.2 to decide when it is the most suitable moment to switch to IDFS phase. We compute an estimated expected number of visited nodes in T_c if the IDFS phase would be started at the given moment:

$$E_1 = k^d |T_{ic}| \textsc{ExpNvn}(|T_c|, p_{ic}, p_c)$$

Then we compute an estimated expected number of visited nodes if one more BFS step would be performed and after that the IDFS phase would be started:

$$E_2 = \textsc{ExpBfs}(T_c, T_v, T_{ic}) + k^{d-1} T_{ic}| \textsc{ExpNvn}(k|T_c|, p_{ic}, p_c)$$

If E_1 is smaller then E_2 we start IDFS phase. We do it only if the lists T_c and T_{ic} are large enough and if the current branching factor is larger than 1. If the list are relatively small (which means that the branching factor is being reduced effectively), the IDFS would slow the search, so we continue the bidirectional search. This enables, in particular, that slowly synchronizing automata in Subsection 5.2 and other automata with strongly reduced branching factor are processed quickly by the algorithm.

Our experiments show that finding a bound by other heuristic algorithms combined with the IDFS shortcut is a really good optimization as it reduces computation time by as much as 83% and 88% for automata with $n = 150$ and $n = 200$ states, respectively.

5 Complexity

The efficiency gain of the algorithm relies mainly on two properties of the majority of automata. First, the average size of subsets decreases fast during the first BFS steps, but increases slow during IBFS steps (cf. Subsection 4.2). Due to this fact the maintained subsets are usually small. Second, the branching factors of both BFS and IBFS are less than k, because of skipping redundant visited sets. Both of the properties are hard to study in a theoretical way, we however have observed them in series of experiments.

In this section we analyze the main part of the algorithm with all optional heuristics, except the sub-tries elimination described in Subsection 4.6. The latter, if applied, worsens theoretical bounds because of more expensive subset-checkings in the worst case.

5.1 Time and space complexity

To provide a theoretical argument we analyze here the expected running time of the algorithm under some artificial assumptions. We give an upper bound for the bidirectional search only, which is a rough estimate of the expected time, but shows a significant impact of the automata properties on performance.

The following assumptions are made:

1. The overall branching factor is r in each step of both BFS and IBFS, $1 < r < k$. This corresponds to an efficient branching factor, which in view of our experiments is considerably less than k.
2. The sets in the tries T_c, T_v and T_{ic}, T_{iv} have random Bernoulli distribution: in each step, they contain any given state with probability $0 < p_c < 1$ (for BFS steps) and $0 < p_{ic} < 1$ (for IBFS steps). We assume also that $p_{ic} \leq p_c$.
3. The steps of BFS and IBFS are performed alternatingly, starting from BFS.
4. No reductions of the visited tries are made and no IDFS phase is performed.

We use RAM computation model in the analysis, with the uniform cost criteria (that is, each elementary operation costs one time unit). We consider r, p_c, p_{ic} as constants and compute a bound as a function of n and k. Let l be the reset length of the automaton. For simplification, assume that l is even.

The initialization phase time may be bounded polynomially by $O(kn^4)$. This includes computing the inverse automaton $O(nk)$, running the heuristic synchronizing algorithms $O(kn^4)$, computing the stationary distribution $O(n^3)$, changing the order of the states of the automaton $O(nk + n \log n)$, and initializing the tries $O(n^2)$.

Under the assumption on the branching factor, the number of sets in T_c in i-th BFS step, after performing $(i-1)$-BFS steps and $(i-1)$-IBFS steps, can be bounded by r^i, which is the number of sets after the step. The number of sets in T_v can be bounded by summing added sets during all the BFS steps: $\sum_{j=0}^{i} r^j = \frac{r^{i+1}-1}{r-1} \in O(r^i)$. Similar bounds hold for T_{ic} and T_{iv}, but there are n sets at the beginning, so it yields $O(nr^i)$.

Recall that under assumptions above we have an estimation for the visited number of nodes in the trie

$$\textsc{ExpNvn}(m, p, q) = \left(\frac{1+p}{p} + \frac{1}{q-pq}\right) m^{\log_w (1+p)},$$

where $w = \frac{1+p}{1+pq-q}$ (cf. Subsection 6.1). Since we we use this formula for various pairs p and q, we shall use an abbreviation $w(p,q) = \frac{1+p}{1+pq-q}$.

Note that each of computing an image or preimage of a set, checking the size of a set, checking if a set is a subset or superset of another set, can be done in $O(n)$ time. Subset checking for one set can be done in expected time $O(n\textsc{ExpNvn}(m,p,q))$, for suitable m,p,q. This is so, because a single visited node costs $O(n)$ if we use the heuristic from Subsection 4.6, and test if the set is a subset of a stored set in leafs, which also costs $O(n)$.

The expected time of the BFS step includes sorting of sets in L (this is done by counting sort, in this case), computing the image of each set by each letter, and checking for visited subsets. So we can bound this by

$$O\left((nr^i) + (knr^i) + (knr^i \textsc{ExpNvn}(O(r^i), p_c, p_c))\right).$$

The last component in the sum is dominating, which yields

$$O\left(knr^i (r^i)^{\log_{w(p_c,p_c)} (1+p_c)}\right).$$

Similarly for the bound for the expected time of IBFS step we obtain:

$$O\left(knr^i (nr^i)^{\log_{w(p_{ic},p_{ic})} (1+p_{ic})}\right).$$

Considering the goal test, it is enough to count only the goal test time after the IBFS step (multiplied by 2). This can be bound by

$$O(nr^i \textsc{ExpNvn}(O(r^i), p_{ic}, p_c)) = O(n^2 r^i (r^i)^{\log_{w(p_{ic},p_c)}(1+p_{ic})}).$$

Computing estimated expected step times after i-th BFS step and i-th IBFS are done in $O(nr^i)$ (having access to list of sets in a trie in linear time), so it may be neglected.

Summing these all yields under domination of the BFS and IBFS step time and the goal test:

$$\begin{aligned}&O\left(knr^i\left((r^i)^{\log_{w(p_c,p_c)}(1+p_c)} + (nr^i)^{\log_{w(p_{ic},p_{ic})}(1+p_{ic})}\right) + n^2 r^i (r^i)^{\log_{w(p_{ic},p_c)}(1+p_{ic})}\right)\\ &\in O\left(kn^2 r^i (r^i)^d\right)\\ &= O\left(kn^2 (r^i)^{1+d}\right)\end{aligned}$$

where

$$d = \max((\log_{w(p_c,p_c)}(1+p_c)), (\log_{w(p_{ic},p_{ic})}(1+p_{ic})), (\log_{w(p_{ic},p_c)}(1+p_{ic}))).$$

The parameter d depends on the distribution of sets in the tries. Note that $0 < d < 1$, so we could bound n^d simply by n.

We can now sum over the steps and obtain as the final result the time complexity:

$$\begin{aligned}&\sum_{i=1}^{l/2} O\left(kn^2 (r^i)^{1+d}\right)\\ &\in O(kn^2\left(\frac{r^{(1+d)(l/2+1)}-1}{r^{1+d}-1}\right))\\ &\in O(kn^2 r^{l(1+d)/2)})\end{aligned}$$

The expected space complexity can be bounded by counting stored sets and nodes in the tries after the last step. There are $O(r^{l/2})$ sets in each of the tries. Each set requires $O(n)$ space, also it induces at most $O(n)$ nodes in a trie. The initialization phase can be done in $O(nk+n^2)$ space. So we can state up the space bound for $O(n(k+n)+nr^{l/2})$. We may summarize these considerations in

Theorem 2 *Under the assumptions (1-4) above, and with l denoting the reset length of the automaton, the expected time complexity of the algorithm is $O(kn^2 r^{l(1+d)/2)})$, and the space complexity is $O(kn^2 + nr^{l/2})$.*

We can observe that the expected time is exponential with regard to the length l, but the exponent is less than l, since $(1+d)/2 < 1$. It is an improvement over the standard BFS algorithm, which has time bound $O(knR^l)$ (assuming we can check visited sets in constant time). Moreover the standard algorithm has usually larger effective branching factor $R \geq r$, since strict supersets of visited sets are not skipped. The expected space complexity yields also an improvement comparing with $O(nR^l)$ space bound for the standard BFS.

For example, for automata with $n = 200$ states on 2 letters we experimentally obtained that the effective branching factor is 1.88 for BFS and 1.36 for IBFS, while the standard BFS has 1.93 in this case. The average sizes of sets are about 0.1 and the corresponding d parameter is about 0.5 for most of steps except a few first, so in this case this would yield to the exponent $0.75l$.

5.2 Performance on slowly synchronizing automata

Let us recall that by the Černý automaton $\mathscr{C}_n$ we mean an n-state automaton on the set $Q = \{0, 1, \ldots, n-1\}$ of states with two input letters a and b such that one letter, say a, satisfies $0a = 1a = 1$, and $xa = x$, otherwise, while the second letter, b acts as the cyclic permutation $xb = x + 1$ (modulo n). It is well known that this automaton has the reset length equal to $(n-1)^2$. In (Ananichev et al, 2012), the authors introduce the series of, what they call, *slowly synchronizing automata* $\mathscr{D}'_n, \mathscr{W}_n, \mathscr{F}_n, \mathscr{E}_n, \mathscr{D}''_n, \mathscr{B}_n, \mathscr{G}_n, \mathscr{H}_n$ with the property that the reset length of these automata is quadratic in terms of the number of states n and close to the Černý bound $(n-1)^2$.

Now, while, generally, our algorithm is exponential in the reset length l, surprisingly, it works fast in polynomial time for all the slowly synchronizing automata defined in (Ananichev et al, 2012). Since the proof is different but very similar in each case, we demonstrate it only for the Černý automata $\mathscr{C}_n$. In other cases the proof is left to the reader on the basis of the definitions given in (Ananichev et al, 2012).

Theorem 3 *For the class of the Černý automata $\mathscr{C}_n$, (and all the classes of slowly synchronizing automata defined in (Ananichev et al, 2012)) the algorithm works in $O(n^4)$ time and $O(n^2 \log n)$ space.*

Proof For each automaton, time used by the preprocessing phase, consisting of the heuristics, reordering the automaton and initializing can be bounded by $O(n^4)$, and space can be bounded by $O(n^2)$. Further, we consider the Černý automata $\mathscr{C}_n$ for $n > 5$.

First, observe that for $\mathscr{C}_n$ after the first step of IBFS there is only one set $\{0, 1\}$ in T_{ic}, because only one state 1 has the preimage of size 2 by a^{-1}. During the next $n-1$ IBFS steps T_{ic} consists of only one set: $\{n-1, 0\}$, $\{n-2, n-1\}, \ldots, \{1, 2\}$, successively, obtained by applying b^{-1}. Then, after the next step the only set in T_{ic} is $\{0, 1, 2\}$. Other preimages are contained in the set that have been already created and are in the visited trie T_{iv}. Generally, T_{ic} still consists of a single set whose size increases by 1 after each consecutive n IBFS steps. Since the cost of single superset checking is $O(n^2)$ (because the size of T_{iv} is $O(n)$ as we argue below), the total cost of the IBFS steps (without reductions of T_{iv}) is $O(n^4)$.

For reductions of T_{iv} we use the fact that after each step there is only one set added to T_{iv}. Generally, the added sets are of the form $\{0, 1, \ldots, i\}, \{n-1, 0, \ldots, i-1\}, \ldots, \{j, j+1, \ldots, j+i\}$ (modulo n). So, after the reduction there are at most n sets in T_{iv}. Since the reduction is performed if T_{iv} has grown $k^2 = 4$ times since the last reduction, we have at most $O(n)$ sets to reduce and a single reduction costs $O(n^3)$. Also there are not more than $O(\log n)$ reductions, so the total cost of reductions is $O(n^3 \log n)$. This yields that the total cost of the IBFS steps is $O(n^4)$.

Consider now the BFS steps. At the beginning we perform some BFS steps due to the reduction of the automaton until $|T_c| > sn$, for some constant s (see Section 4.3). Let us analyze this phase of the algorithm. We start from $Q = \{0, 1, \ldots, n-1\}$ and after the first step we end with the set $Qa = \{1, \ldots, n-1\}$ in T_c, since $Qb = Q$ and it is skipped. After the second step only applying b yields a new set $Qab = \{0, 2, \ldots, n-1\}$. In the next step the set $Qaba = Qa$, so the only new set, and the only member of T_c is $Qabb = \{0, 1, 3, \ldots, n-1\}$. Since the latter set has both 0 and 1, applying both the letters a and b yields new sets. If n is large enough this pattern repeats: after applying a we need to apply b twice to obtain a set containing both 0 and 1, so that applying both the letters a and b yields new sets. This argument may be used to prove that, in general, $|T_c^i| = |T_c^{i-1}| + |T_c^{i-3}|$ for $i < n$, where $|T_c^i|$ is the number of sets after the i-th BFS step. In particular, T_c is growing exponentially. So, for a sufficiently large n there is $O(\log n)$ steps of BFS at the beginning, until $|T_c| > sn$. Under this condition, the size of T_c is

$O(n)$, and the size of T_v is bounded by $O(n \log n)$. Thus, a single step of BFS in this phase costs $O(n^3 \log^2 n)$ and all the steps in this phase cost $O(n^3 \log^3 n)$. Note that no reductions of T_v are performed in this phase. When we get $|T_c| > sn$, then after that only IBFS are performed, since it is $|T_{ic}| \leq sn < |T_c|$ for all the remaining steps.

Finally we must consider the goal tests. A single goal test costs $O(n^2)$, since we have $O(n)$ sets in T_c and $O(1)$ sets in T_{ic}. Summing up all time costs we obtain $O(n^4)$, as required.

Total used space can be bounded by the space used in the preprocessing phase, and by the tries T_c, T_v, T_{ic} and T_{iv} which is $O(n^2)$, $O(n^2 \log n)$, $O(n)$, $O(n^2)$ respectively. Hence the total space is $O(n^2 \log n)$.

As mentioned, the proof for the slowly synchronizing automata defined in (Ananichev et al, 2012) is very similar. The most important difference is the growth factor in BFS steps, but it is exponential in all the cases. In addition, for $\mathscr{D}''_n$,$\mathscr{B}_n$,$\mathscr{F}_n$,$\mathscr{E}_n$,$\mathscr{H}_n$ there can appear two sets in T_{ic} (rather than one), but this is $O(1)$, anyway.

6 Radix tries and subset checking

In this section we analyze the operation of subset checking for radix tries. Recall ((Morrison, 1968)), that a *radix trie* is a binary rooted tree which stores sets in leafs. Each leaf stores one set. We consider radix tries as dynamic data structures that store subsets of a countable universe $U = \{x_1, x_2, \ldots\}$. A set $X \subseteq U$ may be may be identified with the sequence $(b_1, b_2, \ldots)$, where $b_i = 1$ if $x_i \in X$, and $b_i = 0$, otherwise. This sequence determines the unique path in the trie starting from the root to the leaf. At level h, if $x_{h+1} \in X$, then the path goes into the right child, and otherwise it goes into the left child. The path is truncated at the first node which does not belong to any other path induced by another set, and the node becomes a leaf in the trie.

Let m be the number of sets stored in a trie, and let n be the cardinality of the universe. The *insert* procedure for a set S can be performed in time $O(h)$, where h is the maximum height of the trie. This is done by following the path encoded by S from the root to a leaf or to a first node not in the trie. In the latter case, we add the corresponding new node to the trie to store S. In the former, the leaf stores a set S', and we extend the both paths until they diverges. Thus, $h \leq n$. If $n = \infty$, then the path can be arbitrary long. However if we consider random sets, it is known that the average height of a trie is $O(\log(m))$ for any square integrable probability of containing each element in a set (see (Devroye, 1982)); for other consideration concerning parameters of tries for random keys see (Szpankowski, 1991)).

We consider *subset checking* procedure which decides for a given set $S = \{s_1, s_2, \ldots\}$ if there is a subset of S stored in a given trie. The subset checking is performed by a standard DFS search with cut-off determined by the following rule. Starting from the root, at the level h if $s_{h+1} \in S$ the searching goes into the branches of the two children and otherwise it does only into the left child. The right child can be skipped since any subset of S cannot have the $h+1$-th element if $s_{h+1} \notin S$. When a leaf is reached, a simple subset test is performed for S and the stored set.

If a radix trie is used only for insertions and subset checking queries, one may optimize the insert procedure to reduce the number of stored sets in the trie. If the insert procedure for a set S reaches a leaf, and the stored set S' in the leaf contains S, one may replace S' by S, instead of extending the path in order to store both the sets. In that way the number of stored sets in the trie after m insertions can be less than m, but every subset checking will give the same result as if we store both the sets in the trie. For certain distributions it may reduce the size of the trie considerably.

6.1 Bounds for the expected number of visited nodes

We provide here an analysis of the expected time for the subset checking procedure. For the dual version of the procedure, the superset checking, all the results are analogous, one needs only to apply the corresponding dual assumptions (for example, the probability of containing an element should be replaced by the probability of not containing it).

We say that a set $S \subset X$ is a random subset of X with Bernoulli distribution in $[q, r]$ if each element x of X is a member of S with probability $p_x \in [q, r]$, independently of other elements. Given m, we say that a family of subsets of X is a random family of m subsets with Bernoulli distribution in $[q, r]$, if each member of $\mathcal{F}$ is, independently, a random subset of X with Bernoulli distribution in $[q, r]$.

Theorem 4 *Let $p, q, r \in (0, 1)$ be such that $q \le r$ and $q > pr$. Let $\mathcal{F}$ be a random family of m subsets of a given set X with Bernoulli distribution in $[q, r]$, and let S be a random subset of X with Bernoulli distribution in $[0, p]$. Then in the trie constructed for the family $\mathcal{F}$, the expected number of visited nodes by the subset checking procedure for S is at most*

$$\left(\frac{1+p}{p} + \frac{1}{q-pr}\right) m^{\log_w (1+p)},$$

where $w = \frac{1+p}{1+pr-q}$.

Proof Let $f(S, \mathcal{F})$ be a function which counts the visited nodes in the trie constructed for $\mathcal{F}$ by subset checking procedure for S. We may consider it as a random variable, since S and $\mathcal{F}$ are random. Let $\mathbb{E}[f(S, \mathcal{F})]$ be the expected number of visited nodes.

By the definition of the the expected value

$$\mathbb{E}[f(S, \mathcal{F})] = \sum_{S,\mathcal{F}} f(S, \mathcal{F}) \mathrm{P}[S, \mathcal{F}],$$

where the sum is in the probabilistic space over all possible sets S and families $\mathcal{F}$ with m sets, and $\mathrm{P}[S, \mathcal{F}]$ is the probability of occurring S and $\mathcal{F}$. Because choosing S and $\mathcal{F}$ is independent, we have that

$$\mathbb{E}[f(S, \mathcal{F})] = \sum_{S} \sum_{\mathcal{F}} f(S, \mathcal{F}) \mathrm{P}[S] \mathrm{P}[\mathcal{F}].$$

We define the function $f_h(S, \mathcal{F})$ which counts the visited nodes only at the level h in the constructed trie. If h is larger that the cardinality of the universe then $f_h(S, \mathcal{F}) = 0$. So we have:

$$\begin{aligned}
f(S, \mathcal{F}) &= \sum_{h=0}^{\infty} f_h(S, \mathcal{F}) \\
\mathbb{E}[f(S, \mathcal{F})] &= \sum_{S} \sum_{\mathcal{F}} \sum_{h=0}^{\infty} f_h(S, \mathcal{F}) \mathrm{P}[S, \mathcal{F}] \\
&= \sum_{h=0}^{\infty} \sum_{S} \sum_{\mathcal{F}} f_h(S, \mathcal{F}) \mathrm{P}[S, \mathcal{F}] \\
&= \sum_{h=0}^{\infty} \mathbb{E}[f_h(S, \mathcal{F})]
\end{aligned}$$

Consider $\mathcal{F}$ as a subtrie of the complete trie. Then $f_h(S,\mathcal{F})$ can be written as a sum over the nodes in the complete trie at the height h:

$$\sum_{x \text{ at height } h} g(x,S,\mathcal{F}),$$

where $g(x,S,\mathcal{F})$ is an indicator function taking 1 if the node x is visited and 0 otherwise. So we have that

$$\mathbb{E}[f_h(S,\mathcal{F})] = \sum_{x \text{ at height } h} \mathbb{E}[g(x,S,\mathcal{F})].$$

We will estimate now probability that a node is visited at the height h. Let x be a node in the complete trie with the path from the root with exactly i ones and $h-i$ zeros. The node is visited if and only if (1) the searching procedure for a subset of S would reach the node in the complete trie (containing all possible sets) and (2) the node belongs to the constructed trie. These two conditions are independent, since (1) depends only on S and (2) only on $\mathcal{F}$. We may define therefore two indicator functions $g'(x,S)$ which takes 1 if and only if the first condition holds and $g''(x,\mathcal{F})$ which takes 1 if and only if the second condition holds.

We bound the probability that condition (1) holds. It holds if and only if S contains all the elements corresponding to ones in the path (otherwise the search does not go into the corresponding branch). Since the probability of containing each element is in $[0,p]$, the probability that the condition (1) holds does not exceed p^i. Similarly we bound the probability that condition (2) holds. It holds only if there exists a set in $\mathcal{F}$ whose first h elements correspond to the path of the node (in fact, this condition is necessary, but not sufficient, because of truncating paths). The probability that a single subset has a required sequence of the first h elements, with exactly i ones and $h-i$ zeros, in view of the assumption on Bernoulli distribution in $[q,r]$, can be bounded from above by $r^i(1-q)^{h-i}$. Since $\mathcal{F}$ contains m elements, the probability that condition (2) holds may be upper bounded by $\min\{1, mr^i(1-q)^{h-i}\}$. Summarizing, for a node x with i ones and $h-i$ zeros on the path we have:

$$\begin{aligned}
g(x,S,\mathcal{F}) &= g'(x,S)g''(x,\mathcal{F}) \\
\mathbb{E}[g'(x,S)] &= \sum_{S} \mathrm{P}[S]\mathrm{P}[S \text{ has specified } i \text{ elements}] \le p^i \\
\mathbb{E}[g''(x,\mathcal{F})] &= \sum_{\mathcal{F}} \mathrm{P}[\mathcal{F}]\mathrm{P}[\mathcal{F} \text{ has a set with } i \text{ and without } h-i \text{ specified elements}] \\
&\le \min\{1, mr^i(1-q)^{h-i}\}
\end{aligned}$$

Now we can group the nodes at the height h, which have the same number of ones on the path and we can sum over these groups of the nodes, obtaining:

$$\mathbb{E}[f_h(S,\mathcal{F})] = \sum_{x \text{ at height } h} \mathbb{E}[g(x,S)g(x,\mathcal{F})] \le \sum_{i=0}^{h} \binom{h}{i} p^i \min\{1, mr^i(1-q)^{h-i}\}.$$

This yields two bounds that will be used to estimate

$$\mathbb{E}[f(S,\mathcal{F})] = \sum_{h=0}^{\infty} \mathbb{E}[f_h(S,\mathcal{F})].$$

Let $t = \lfloor \log_w m \rfloor$, where $w = \frac{1+p}{1+pr-q}$. We will split up the sum above into two parts: the first one that sums over the levels from 0 to t, and the second one that sums from $t+1$ to n.

Case 1. We estimate $\sum_{h=0}^{t} \mathbb{E}[f_h(S,\mathcal{F})]$. For $g(x,\mathcal{F})$ we use in this case the trivial bound $g(x,\mathcal{F}) \leq 1$. So, we have

$$\mathbb{E}[f_h(S,\mathcal{F})] \leq \sum_{i=0}^{h} \binom{h}{i} p^i = (1+p)^h,$$

and consequently,

$$\sum_{h=0}^{t} \mathbb{E}(f_h(S,\mathcal{F})) \leq \sum_{h=0}^{t} (p+1)^h = \frac{(1+p)^{t+1}-1}{p}.$$

Substituting $t = \lfloor \log_w m \rfloor$ yields

$$\begin{aligned}
\frac{(p+1)^{t+1}-1}{p} &= \frac{(1+p)^{\lfloor \log_w m \rfloor + 1} - 1}{p} \\
&\leq \frac{(1+p)(p+1)^{\log_w(m)} - 1}{p} \\
&= \frac{(1+p)m^{\log_w(1+p)} - 1}{p} \\
&< \frac{(1+p)}{p} m^{\log_w(1+p)}
\end{aligned}$$

Case 2. We estimate $\sum_{h=t+1}^{n} \mathbb{E}[f_h(S,\mathcal{F})]$. For this case we have

$$\mathbb{E}[f_h(S,\mathcal{F})] \leq \sum_{i=0}^{h} \binom{h}{i} p^i m r^i (1-q)^{h-i} = m(1+pr-q)^h,$$

and consequently,

$$\begin{aligned}
\sum_{h=t+1}^{\infty} \mathbb{E}[f_h(S,\mathcal{F})] &\leq \sum_{h=t+1}^{\infty} m(pr-q+1)^h \\
&= \sum_{h=0}^{\infty} m(1+pr-q)^{h+t+1} \\
&\leq \sum_{h=0}^{\infty} m(1+pr-q)^{h+\log_w m} \\
&= \sum_{h=0}^{\infty} m\left(m^{\log_w(1+pr-q)}(1+pr-q)^h\right) \\
&= m^{\log_w(w)+\log_w(1+pr-q)} \sum_{h=0}^{\infty} (1+pr-q)^h \\
&\quad \text{(note that } (1+pr-q) < 1 \text{, since by assumption } q > pr\text{)} \\
&< m^{\log_w((1+pr-q)w)} \frac{1}{q-pr} \\
&= \frac{1}{q-pr} m^{\log_w(1+pr-q)\frac{1+p}{1+pr-q}} \\
&= \frac{1}{q-pr} m^{\log_w(1+p)}
\end{aligned}$$

Combining both the cases we obtain

$$E[f(S,\mathcal{F})] < \frac{1+p}{p} m^{\log_w(p+1)} + \frac{m^{\log_w(1+p)}}{q-pr} = \left(\frac{1+p}{p} + \frac{1}{q-pr}\right) m^{\log_w(1+p)},$$

as required.

The bound of the theorem is essentially better for certain distributions than the trivial $O(m)$ bound, and this seems to be one of the major reasons for which our algorithm is so efficient.

If m is very large relative to n, it is possible to get a better (and simpler) bound:

Theorem 5 *Consider sets whose elements are from a finite n-element universe. Let S be a random set having $[0,p]$-bounded distribution. Then in the trie constructed for any family $\mathcal{F}$ the expected number of visited nodes by the subset checking procedure for S is at most*

$$\frac{(p+1)^n - 1}{p},$$

and the bound is tight for some $\mathcal{F}$.

Proof We follow the proof of 4, but we set $t = n-1$. We only need considering the first case:

$$\sum_{h=0}^{t} \mathbb{E}[f_h(S,\mathcal{F})] \le \sum_{h=0}^{n-1} (1+p)^h = \frac{(1+p)^n - 1}{p}.$$

The bound is tight, since we can get the complete family $\mathcal{F}$ containing all possible sets, and S containing each element with probability exactly p. Then we can observe that we can follow only by equalities for $E(f(S,\mathcal{F}))$, since $g(x,S) = p^i$ and $g(x,\mathcal{F}) = 1$, so $\sum_{h=0}^{t} \mathbb{E}[f_h(S,\mathcal{F})] = \sum_{h=0}^{n-1}(1+p)^h$ in this case.

This leads immediately to the following corollary on asymptotic:

Corollary 1 *If n is fixed, S contains each element with probability p independently and each possible set has non-zero probability of being in $\mathcal{F}$ then:*

$$\lim_{m\to\infty} \mathbb{E}[S,\mathcal{F}] = \frac{(1+p)^n - 1}{p}$$

This shows another reason, in addition to the function shape of the upper bound from Theorem 4, why the distribution of queried sets is more important than that of stored sets. Therefore our heuristic techniques such us reordering of the automaton states (4.4) prefer optimizing the former even at the cost of worsening the distribution of stored sets.

7 Experiments

We perform a series of the following experiments for various $n \le 350$. For a given n, we generate a random automaton A with n states and 2 input letters, check whether A is synchronizing and if so, we find the reset length using the algorithm described above. On the basis of the obtained results we estimate the expected reset length. Then we have performed similar experiments for automata with $k = 3, \ldots, 10$ input letters.

Table 1 The comparison of average computation time and the maximum time for random automata.

n	50	100	150	200	250	300	350
Standard BFS/DFS	0.019 s	10.047 s	8 min 5 s	–	–	–	–
Our average time	0.005 s	0.021 s	0.13 s	1.09 s	8.24 s	55.74 s	6 min 28 s
Our maximum time	0.03 s	1.45 s	3.48 s	67.63 s	418.65 s	150 min 5 s	25 h 7 min

7.1 Computations

In the experiments we have used the standard model of random automata, where for each state and each letter all the possible transitions are equiprobable. A random automaton with n states and k input letters can be then represented as a sequence of kn uniformly random natural numbers from $[0, n-1]$. To generate high quality random sequences we have used the WELL number generator (Panneton et al, 2006) (variants 1024 and 19937) seeded by random bytes from Unix `/dev/random` device. For $k = 2$ input letters, we have computed exact results for automata up to 7 states by checking all of them; for each $8 \leq n \leq 100$ we have checked one million automata, and for each $101 \leq n \leq 250$ and $n = 255, 260, \ldots, 350$ we have checked $10,000$ automata.

The computations have been performed mostly on 16 computers with Intel(R) Core(TM) i7-2600 CPU 3.40GHz 4 cores and 16GB of RAM. The algorithm was implemented in C++ and compiled with g++. Distributed computations were managed by a dedicated server and clients applications written in Python.

7.2 Efficiency

The average computation time is about 100 up to 1000 times faster than the time of Trahtman's program TESTAS (Trahtman, 2003, 2006) for binary automata with 50 states. The reduction to SAT used in (Skvortsov and Tipikin, 2011) seemed to be the fastest recently known algorithm and the reported average time for 50 states automata is 2.7 seconds, and for 100 states automata is 70 seconds. Our comparable results are less than 0.005 and 0.021 seconds, respectively (we have used faster machines but the supplementary resulting speed-up should be not more than by 2 times).

Table 1 presents the comparison of average computation times for our algorithm and the standard BFS method. The programs were run with 16 GB limited memory. Standard BFS/DFS is our optimized implementation of the standard BFS algorithm in the power automaton, which additionally switch to DFS (analogous to Algorithm 4) when it runs out of memory. The BFS alone was unable to process automata with 100 states or more due to its memory requirements.

The average times are relatively small because of rare occurrences of automata with long reset words. So we present also the maximum computation time which is much larger than the average one since it depends on the automaton generated. Our experiment did not find any really "hard" example.

7.3 General results

Our experiment confirms that for the standard random automata model $\mathbb{A}(n)$ the probability that the automaton is synchronizing tends to 1 as the number n of states grows (this conjecture posed in (Skvortsov and Tipikin, 2011) has been verified recently by Berlinkov (2013)). In our experiment, for $n = 100$, 2250 of one million automata turned out to be non-synchronizing

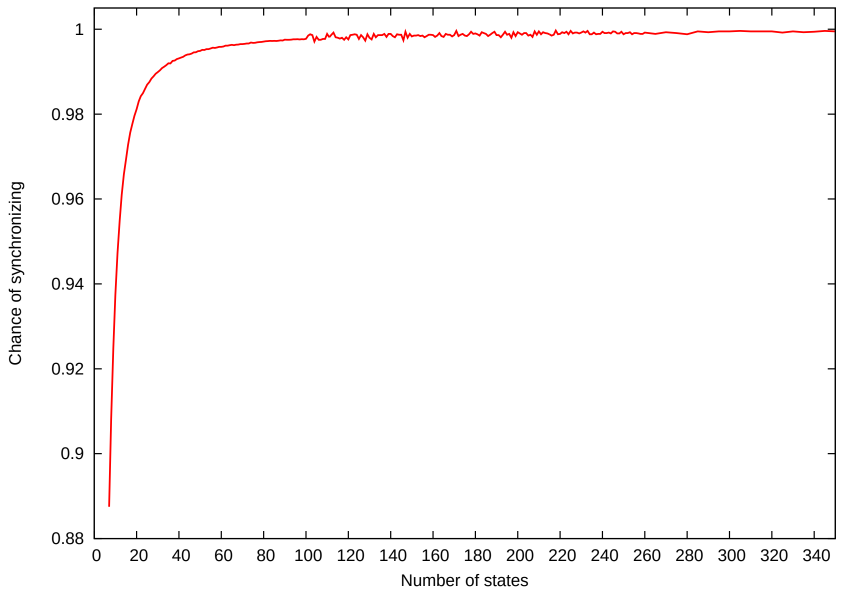

Fig. 1 Experimental values of synchronization probability.

(0.225%), and for $n = 300$, only five of $10,000$ automata. For automata with 2 input letters and up to 100 states the line of synchronization chance is presented in Figure 1.

We observe also that random automata are mostly not strongly connected. Moreover, as mentioned in Subsection 4.3, if an automaton is synchronizing then the expected size of the strongly connected sink component seems to tend to the value $\approx 0.7987n$. We also noted that the average length of the minimal synchronizing word in a random automaton is usually a little larger than the length in the strongly connected automaton formed by its sink component.

7.4 The expected reset length

The main result of the experiments is the estimation of the expected minimal length of a synchronizing word of an automaton A. We consider the infinite sequence of random variables $\ell(n)$ defined as the reset length for a synchronizing automaton with n states. By $\mathbb{E}[\ell(n)]$ we denote the expected value of $\ell(n)$, and by $\mathbb{V}[\ell(n)]$ its variance. Let $ML(n)$ denotes the mean of reset lengths of the automata with n states generated in our experiment.

In (Skvortsov and Tipikin, 2011), the authors assume that $ML(n)$ is a good approximation of $\mathbb{E}[\ell(n)]$. Usually, it is the Hoeffding's inequality that is used to estimate how good is this approximation. Unfortunately, the number of experiments performed in (Skvortsov and Tipikin, 2011) is far too small to make use of this inequality.

In contrast, our experiments allow to obtain a good estimation of the approximation error. We have the following:

Theorem 6 *Let M be the maximal reset length in the sample of m randomly generated automata from the class $\mathcal{A} = \mathcal{A}(n)$ of the synchronizing n-state automata. If r is the fraction of automata in $\mathcal{A}$ having the reset length larger than M, then with probability $1 - p$*

$$|ML(n) - \mathbb{E}[\ell(n)]| \leq (1-r)M\sqrt{\frac{\log(2/p)}{2m}} + r\frac{n^3 - n}{6}.$$

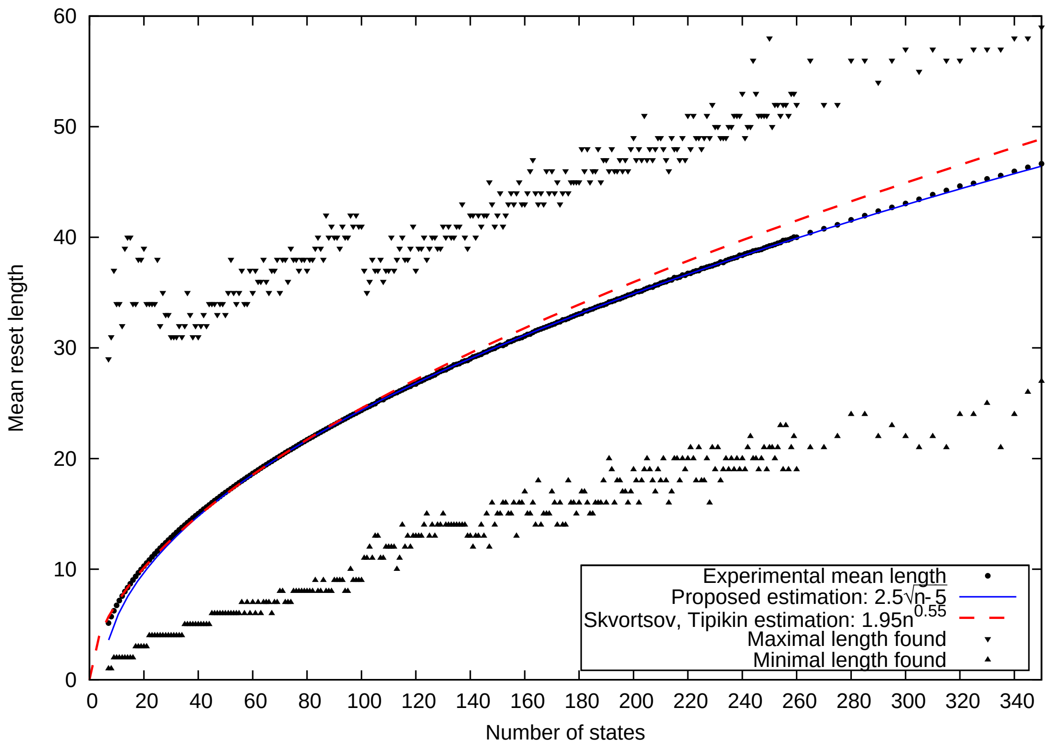

Fig. 2 The approximations proposed for the expected reset length.

Proof We make use of the Hoeffding's inequality (Hoeffding, 1963) given in the following form. For $0 < p \leq 1$, with probability at least $1 - p$

$$|\overline{X} - \mathbb{E}[\overline{X}]| \leq R\sqrt{\frac{\log(2/p)}{2m}}, \tag{1}$$

where $\overline{X} = (X_1 + \ldots + X_m)/m$ is the empirical mean of random variables $X_1, \ldots, X_m$ with the same range R.

Since the distribution of $\ell(n)$ is highly asymmetric, one needs to combine this inequality with the statistical fact that the maximal reset lengths obtained in the experiment are much smaller than the known bounds and that larger lengths occur rarely.

Let r denotes the fraction of automata in $\mathcal{A}$ having the reset length larger than M, for any fixed $M > 0$. First we assume that we sample only automata with the reset length $\leq M$. Denote the corresponding random variable by $\ell'(n)$. Applying the Hoeffding's inequality, putting $X_1 = \ldots = X_m = \ell'(n)$, and $R = M$ we obtain

$$|ML(n) - \mathbb{E}[\ell']| \leq M\sqrt{\frac{\log(2/p)}{2m}}.$$

Let $\ell''(n)$ be the reset length for a synchronizing automata with n states and $\ell(n) \geq M$. Then we obtain

$$|ML(n) - \mathbb{E}[\ell]| \leq (1-r)|ML(n) - \mathbb{E}[\ell']| + r|ML(n) - \mathbb{E}[\ell'']| \leq (1-r)M\sqrt{\frac{\log(2/p)}{2m}} + r\frac{n^3}{6}.$$

We have used the well-known bound $\frac{n^3-n}{6}$ for the length of the shortest reset word. Taking M to be the maximal reset length of the automata in the sample, we obtain the required result.

Assuming the Černý conjecture in the last term $(n^3 - n)/6$ may be replaced by $(n - 1)^2$ (giving essentially better estimation).

For $n = 100$, $m = 10^6$, we have obtained $ML(n) \approx 24.34$, and the maximal reset length $M = 41$. If the fraction of those automata in $\mathcal{A}$ with the reset length exceeding $M = 41$ is greater than $r = 0.00001$, the probability than no automaton was generated with the reset length > 41 is less than $q = (1 - r)^m = 0.00005$. So we may assume that with a very high probability $r < 0.00001$. Now, for $p = 0.00005$, it follows that with probability $1 - p = 0.99995$

$$|ML(n) - \mathbb{E}[\ell(n)]| \leq 0.0943 + 1.666 < 1.68,$$

which means that the error is less than 1.68 (or 0.19 assuming the Černý conjecture). This means that with high probability the expected length of the shortest reset word for synchronizing automata with $n = 100$ states is close to our experimental result 24.34. Comparing this with the results of Skvortsov and Tipikin (Skvortsov and Tipikin, 2011), we note that, for automata with 100 states, they also have obtained the expected length close to 24, but taking into account the size of the sample $m = 200$, no reasonable estimation of the error can be obtained in this way (even values of p as large as $p = 0.1$ lead to a few hundred percent error).

7.5 New approximation

We have observed that the approximation of the mean length $ML(n) \approx 1.95n^{0.55}$ proposed in (Skvortsov and Tipikin, 2011) is inflated. We have been searching for an approximation function by filling some predefined templates with different constants and comparing them by minimizing the sum of squares of differences with the experimentally computed estimation. Based on currently available data, we propose a new more precise experimental approximation for the expected reset length for automata with 2 input letters. Note, that our approximation below is supported also by Theorem 6.

$$\mathbb{E}[\ell(n)] \approx 2.5\sqrt{n-5}. \tag{2}$$

Comparison of both the proposed functions with the experimental results is presented in the Figure 2. The dashed line is the approximation proposed by Skvortsov and Tipikin, while our approximation is covered almost exactly by dots representing experimental results. Small triangles above and below two lines represent, respectively, the maximal and minimal reset lengths found. We observe that the expected length seems to belong to $\Theta(\sqrt{n})$ anyway.

7.6 Distribution and variance

The results of our experiment allow to compute an approximated probability distribution of $\ell(n)$ for each tested n. Example distributions are shown in Figure 3. They are very similar for larger n. For $n = 7$ states the exact distribution is presented.

We have also confirmed the observations from (Skvortsov and Tipikin, 2011) that the variance $\mathbb{V}[\ell(n)]$ is a growing function. We however do not confirm that the fraction $\frac{\sqrt{\mathbb{V}[\ell(n)]}}{\mathbb{E}[\ell(n)]}$ seems to tend to 0 as n goes to infinity. The graph we have obtained (Figure 4) does not exclude the possibility that the fraction converges to some positive constant.

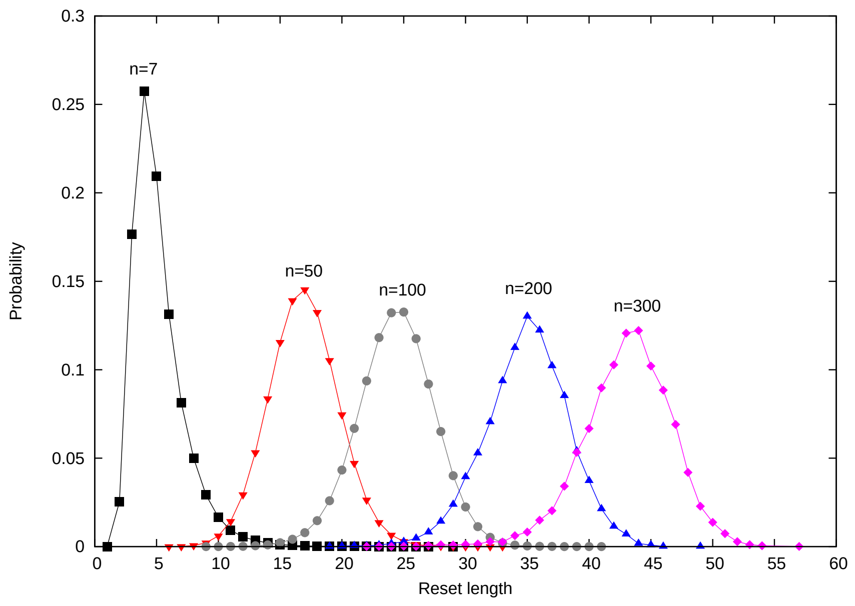

Fig. 3 Distributions of $\ell(n)$ for various n.

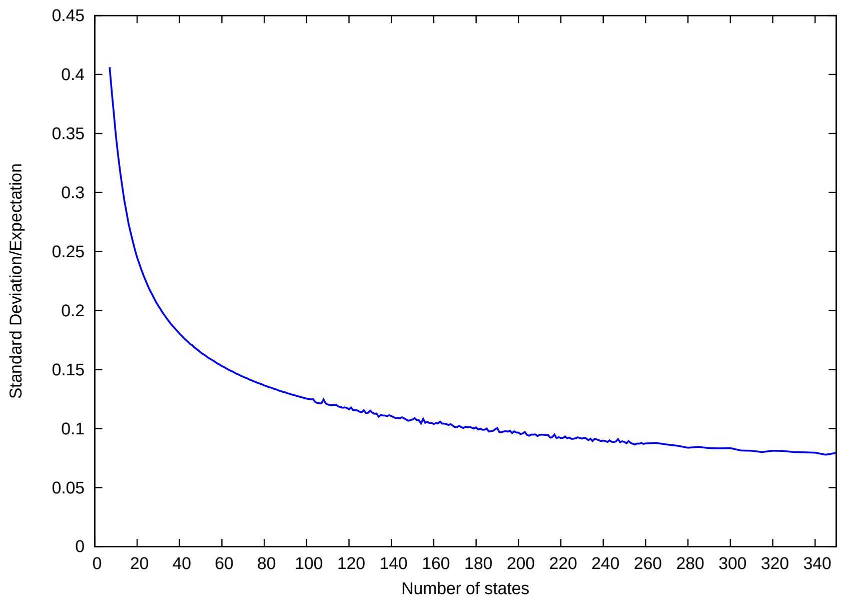

Fig. 4 Graph of $\frac{\sqrt{\mathbb{V}[\ell(n)]}}{\mathbb{E}[\ell(n)]}$.

7.7 Automata with more than 2 input letters

We have performed also a series of experiments to see if and how the situation changes in case of automata with more than 2 input letters. The results shows that certain trends observed for

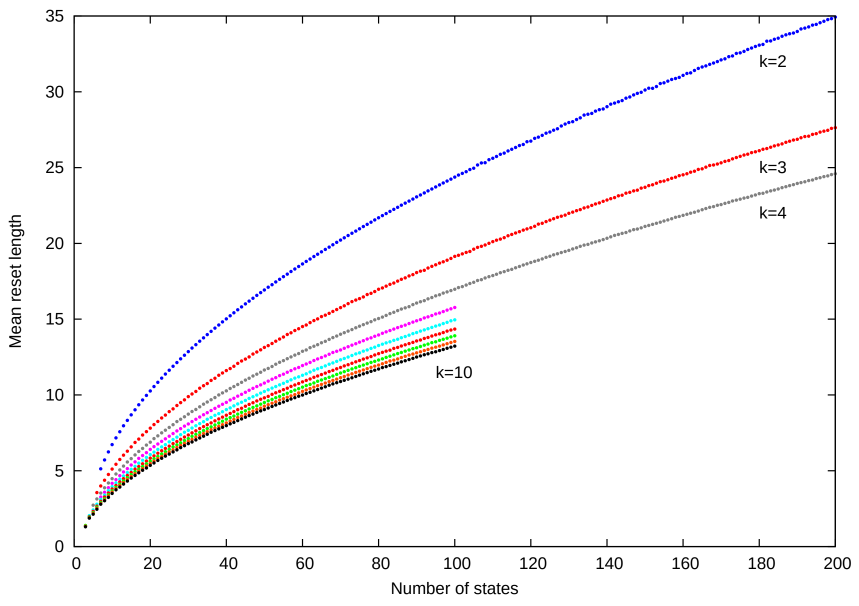

Fig. 5 Mean reset length for synchronizing automata with k input letters.

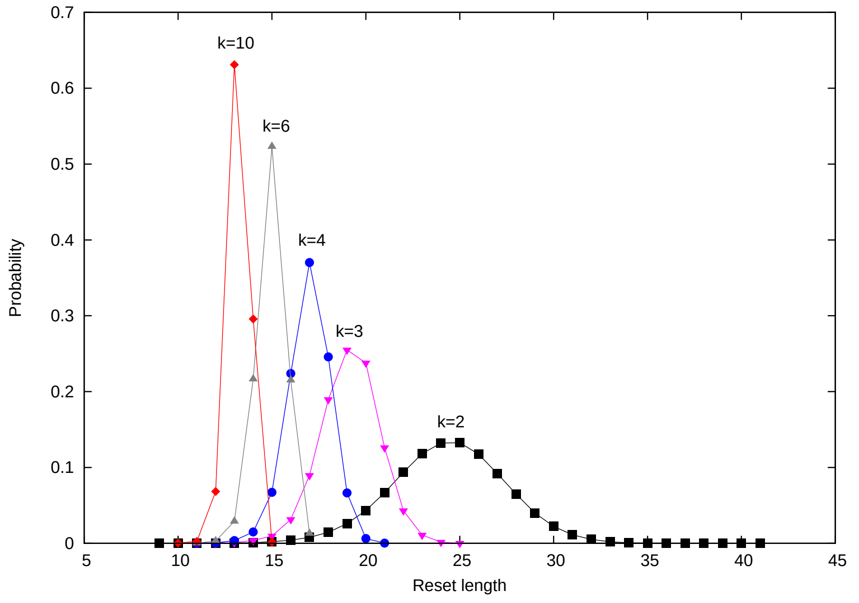

Fig. 6 Distributions of the reset length for automata with $n = 100$ states and k input letters.

automata with 2 letters continue in a regular way. We have used random samples of 10,000 automata for each presented number of states.

The mean reset length decreases when k increases, but the corresponding graphs have similar regular shape. The results of our experiments are pictured in Figure 5. In Figure 6 distributions of the reset length are presented for automata with $n = 100$. For better visibility only results for $k = 2, 3, 4, 6, 10$ are presented. They show clearly the trend of decreasing intervals of length

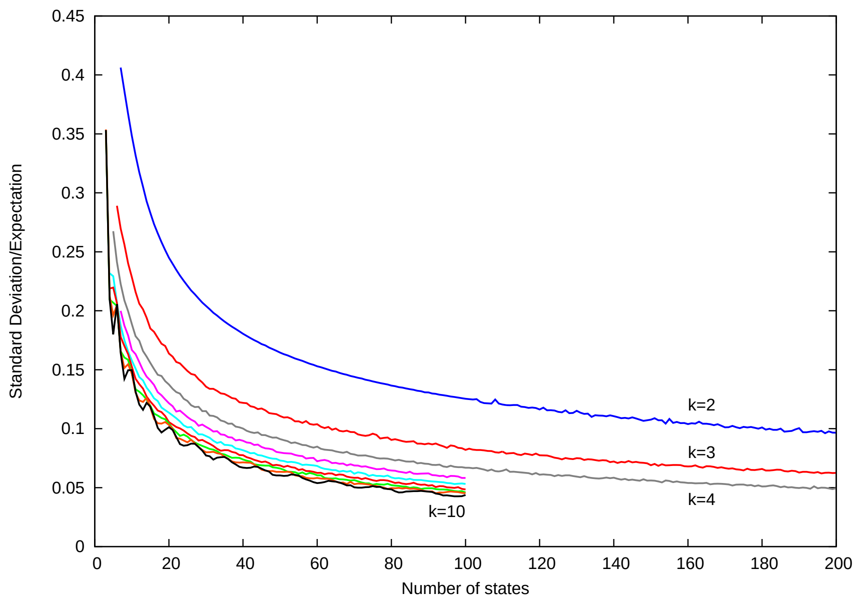

Fig. 7 Relative standard deviations of reset length for automata with k input letters.

values with higher probabilities. Finally, Figure 7 shows relative standard deviations $\frac{\sqrt{\mathbb{V}[\ell(n)]}}{\mathbb{E}[\ell(n)]}$ for automata with k input letters. Again the shape of graphs is similar.

In conclusion, we may say that our experiments did not show any differences in behavior of automata depending on the size of the input alphabet, except for the expected fact that reset lengths decrease as the size increase.

References

Ananichev D, Volkov M (2003) Synchronizing monotonic automata. In: Developments in Language Theory, LNCS, vol 2710, pp 111–121

Ananichev D, Gusev V, Volkov M (2010) Slowly synchronizing automata and digraphs. In: Mathematical Foundations of Computer Science 2010, LNCS, vol 6281, pp 55–65

Ananichev D, Gusev V, Volkov M (2012) Primitive digraphs with large exponents and slowly synchronizing automata. Zapiski Nauchnyh Seminarov POMI [Kombinatorika i Teorija Grafov IV] 402:9–39, in Russian

Batsyn M, Goldengorin B, Maslov E, Pardalos PM (2013) Improvements to MCS algorithm for the maximum clique problem. Journal of Combinatorial Optimization pp 1–20

Benenson Y, Adar R, Paz-Elizur T, Livneh Z, Shapiro E (2003) DNA molecule provides a computing machine with both data and fuel. Proceedings of the National Academy of Sciences 100(5):2191–2196

Berlinkov M (2010) Approximating the minimum length of synchronizing words is hard. In: Computer Science – Theory and Applications, LNCS, vol 6072, pp 37–47

Berlinkov M (2013) On the probability to be synchronizable. http://arxiv.org/abs/1304.5774

Broy M, Jonsson B, Katoen JP, Leucker M, Pretschner A (2005) Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science). Springer-Verlag New York, Inc.

Černý J (1964) Poznámka k homogénnym eksperimentom s konečnými automatami. Matematicko-fyzikálny Časopis Slovenskej Akadémie Vied 14(3):208–216, in Slovak

Chmiel K, Roman A (2011) COMPAS - A computing package for synchronization. In: Implementation and Application of Automata, LNCS, vol 6482, pp 79–86

Devroye L (1982) A note on the average depth of tries. Computing 28:367–371

Eppstein D (1990) Reset sequences for monotonic automata. SIAM Journal on Computing 19:500–510

Gerbush M, Heeringa B (2011) Approximating minimum reset sequences. In: Implementation and Application of Automata, LNCS, vol 6482, pp 154–162

Higgins P (1988) The range order of a product of i-transformations from a finite full transformation semigroup. Semigroup Forum 37:31–36

Hoeffding W (1963) Probability inequalities for sums of bounded random variables. J Amer Statist Assoc 58(301):13–30

Jürgensen, H (2008) Synchronization. Information and Computation 206(9-10):1033–1044

Kari J (2002) Synchronization and stability of finite automata. Journal of Universal Computer Science 8(2):270–277

Kisielewicz A, Szykuła M (2013) Generating small automata and the Černý conjecture. In: Implementation and Application of Automata, LNCS, vol 7982, pp 340–348

Kisielewicz A, J K, M S (2013) A Fast Algorithm Finding the Shortest Reset Words. In: Computing and Combinatorics, LNCS, vol 7936, pp 182–196

Kowalski J, Szykuła M (2013) A new heuristic synchronizing algorithm. http://arxiv.org/abs/1308.1978

Kudłacik R, Roman A, Wagner H (2012) Effective synchronizing algorithms. Expert Systems with Applications 39(14):11,746–11,757

Martyugin P (2009) Complexity of problems concerning reset words for some partial cases of automata. Acta Cybernetica 19:517–536

Martyugin P (2011) Complexity of problems concerning reset words for cyclic and Eulerian automata. In: Implementation and Application of Automata, LNCS, vol 6807, pp 238–249

Morrison D (1968) PATRICIA – practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM 15:514–534

Olschewski J, Ummels M (2010) The complexity of finding reset words in finite automata. In: Mathematical Foundations of Computer Science 2010, LNCS, vol 6281, pp 568–579

Panneton F, L'Ecuyer P, Matsumoto M (2006) Improved long-period generators based on linear recurrences modulo 2. ACM Transactions on Mathematical Software 32(1):1–16

Podolak IT, Roman A, Jędrzejczyk D (2012) Application of hierarchical classifier to minimal synchronizing word problem. In: Artificial Intelligence and Soft Computing, LNCS, vol 7267, pp 421–429

Rho JK, F S (1993) Minimum length synchronizing sequences of finite state machine. In: Proceedings of the 30th ACM/IEEE Design Automation Conference, DAC '93, pp 463–466

Roman A (2009a) Genetic algorithm for synchronization. In: Language and Automata Theory and Applications, LNCS, vol 5457, pp 684–695

Roman A (2009b) Synchronizing finite automata with short reset words. In: Applied Mathematics and Computation, ICCMSE-2005, vol 209, pp 125–136

Sandberg S (2005) Homing and synchronizing sequences. In: Model-Based Testing of Reactive Systems, LNCS, vol 3472, pp 5–33

Skvortsov E, Tipikin E (2011) Experimental study of the shortest reset word of random automata. In: Implementation and Application of Automata, LNCS, vol 6807, pp 290–298

Stewart WJ (1994) Introduction to the Numerical Solution of Markov Chains. Princeton University Press

Stewart WJ (2000) Numerical methods for computing stationary distributions of finite irreducible markov chains. In: Computational Probability, vol 24, pp 81–111

Szpankowski W (1991) On the height of digital trees and related problems. Algorithmica 6(1-6):256–277

Tarjan R (1972) Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2):146–160

Trahtman AN (2003) A package TESTAS for checking some kinds of testability. In: Implementation and Application of Automata, LNCS, vol 2608, pp 228–232

Trahtman AN (2006) An efficient algorithm finds noticeable trends and examples concerning the Černý conjecture. In: Mathematical Foundations of Computer Science, LNCS, vol 4162, pp 789–800

Volkov M (2008) Synchronizing automata and the Černý conjecture. In: Language and Automata Theory and Applications, LNCS, vol 5196, pp 11–27

Xue G, Sun S, Du DHC, Shi L (2000) An Efficient Algorithm for Delay Buffer Minimization. Journal of Combinatorial Optimization 4(2):217–233