# A Fast Algorithm Finding the Shortest Reset Words 

Andrzej Kisielewicz ${ }^{\star 1,2}$, Jakub Kowalski ${ }^{1}$, and Marek Szykuła ${ }^{1}$<br>1 Department of Mathematics and Computer Science, University of Wrocław, Poland<br>${ }^{2}$ Institute of Mathematics and Computer Science, University of Opole, Poland andrzej.kisielewicz@math.uni.wroc.pl, \{kot,msz\}@ii.uni.wroc.pl


#### Abstract

In this paper we present a new fast algorithm for finding minimal reset words for finite synchronizing automata, which is a problem appearing in many practical applications. 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. Also a number of heuristics are applied. 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 $n \leq 300$ states and 2 input letters. In particular, we obtain a new estimation of the expected length of the shortest reset word $\approx 2.5 \sqrt{n-5}$.


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 $\Sigma$, and the transition function $\delta: Q \times \Sigma \rightarrow Q$. The action of $\Sigma$ on $Q$ given by $\delta$ is denoted simply by concatenation: $\delta(q, a)=q a$. This action extends naturally to the action $q w$ of words for any $w \in \Sigma^{*}$. If $|Q w|=1$, 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 robotics (for designing so-called part orienters) [2, bioinformatics (the reset problem) [3, network theory [12], theory of codes [11] etc.

Theoretical research in the area is mainly motivated 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 [4], and is considered the most longstanding open problem in the combinatorial theory of finite automata. So far, the conjecture has been proved only for a few special classes of automata and a general cubic upper bound $\left(n^{3}-n\right) / 6$ has been established (see Volkov [25] for an excellent survey of the results, and Trahtman [24] for a recently found new cubic bound). Using computers the conjecture has been verified for small automata with 2 letters and $n \leq 10$ states (and with $k \leq 4$ letters and $n \leq 7$ states [23]; see also [1] 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

[^0]of the shortest reset word is $\mathrm{FP}^{\mathrm{NP}[\log ]}$-complete, and the related decision problem is both NPand coNP-hard 15 .

On the other hand, there are several theoretical and experimental results showing that most synchronizing automata have relatively short reset words and those slowly synchronizing (with the shortest reset words of quadratic length) are rather exceptional [1. An old result by Higgins [9] on products in transformation semigroups shows that a random automaton with an alphabet of size larger than $2 n$ has, with high probability, a reset word of length $\leq 2 n$. More recently, it was proved that, for every $\epsilon>0$, a random automaton with $n$ states over an alphabet of size $n^{0.5+\epsilon}$, with high probability, is synchronizing and satisfies the Černý conjecture 21. In computing reset words, either exponential algorithms finding the shortest reset words [20|23|13] or polynomial heuristics finding relatively short reset words [8]13|17|18|23] 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 $19|23| 13 \mid 25$. 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 [22] and recently developed COMPAS [5]). In [18], Roman uses a genetic algorithm to find a reset word of randomly generated automata and thus obtains upper bounds on the length of the shortest reset word.

A new interesting approach for finding the exact length using a SAT-solver has been applied recently by Skvortsov and Tipikin [20]. 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 $[1,100]$ 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 [18 that the mean length of the shortest reset word for a random $n$-state synchronizing automaton is linear and almost equal to $0.486 n$. Skvortsov and Tipikin argue that their experiment based on a larger set of data shows that this length is actually sublinear and $\approx 1.95 n^{0.55}$.

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 [14]) are used. 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 one million automata for each $n \leq 100$, (compared with 200-2000 generated by Skvortsov and Tipikin), and we were able to test much larger automata with up to $n=320$ states. Our data confirm the hypothesis that the expected length of the shortest reset word 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 [20.

Our algorithm makes also possible to find a reset word of the shortest length (not only the length). Curiously, it works in polynomial time for known slowly synchronizing automata series [1]. So far, most of the empirical research in the area concerns automata with 2 input letters. Some researches suggest that automata with more letters may exhibit a different behavior. We plan to use the algorithm to perform an extensive research on automata with $k>2$ letters.

## 2 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 [7]. 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 $\Sigma$ (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 refereed to as IBFS). Both the searches have branching factor $k$ (the number of input letters) and need to compute $O\left(k^{l}\right)$ sets (or $O\left(n k^{l}\right)$ 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 in only $O\left(n k^{l / 2}\right)$ steps. 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 General Ideas

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 $Q u$ is a subset of the preimage $\{q\} w^{-1}$ for some $q \in Q$. Consequently, $Q u w=\{q\}$, 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 $|T w|=1$ implies $|S w|=1$ 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) [14]. 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 3 ,

### 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 node at depth $n$ 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. [14].

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 2.3

```
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.
                                    \(\triangleright\) Initialize four radix tries to store and handle subsets of \(Q\) :
    \(T_{c} \leftarrow\) EmptyTrie \(\quad \triangleright\) BFS current trie
    \(T_{v} \leftarrow\) EmptyTrie \(\quad \triangleright\) BFS visited trie
    \(T_{i c} \leftarrow\) EmptyTrie \(\quad \triangleright\) IBFS current trie
    \(T_{i v} \leftarrow\) EmptyTrie \(\quad \triangleright\) IBFS visited trie
    \(T_{c}\).INSERT \((Q)\)
    \(T_{v} \cdot \operatorname{INSERT}(Q)\)
    for all \(q \in Q\) do
        \(T_{i c} \cdot \operatorname{INSERT}(\{q\})\)
        \(T_{i v}\).INSERT \((\{q\})\)
    end for
    for \(l \leftarrow 1\) to maxlen do
        if estimated time of the BFS step is smaller than that of IBFS then
            BFS_STEp \(\left(T_{c}, T_{v}\right)\)
                                \(\triangleright\) Modify BFS tries; minimize \(T_{c}\) using \(T_{v}\)
        else
            IBFS_STEP \(\left(T_{i c}, T_{i v}\right) \quad \triangleright\) Modify IBFS tries; minimize \(T_{i c}\) using \(T_{i v}\)
        end if
        for all \(S \in T_{i c}\) do \(\triangleright\) The goal test loop
            if \(T_{c}\).CONTAINS_SUBSET_OF \((S)\) then
                return \(l \quad \triangleright\) The length of the shortest reset word
                end if
        end for
    end for
    return "No synchronizing word of length \(\leq\) maxlen"
```


### 2.3 Description

The main part of the algorithm is given in Algorithm 1. To make it clearer we restrict the task to finding the shortest length of a reset word only. Yet, the algorithm can be easily modified to return also a reset word of such length (see 2.4).

We use, in principle, four radix tries $T_{c}, T_{v}, T_{i c}, T_{i v}$ 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 3.1

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 $T_{i c}$, 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 shortest length of a rest word for $A$. To prove this we need to analyze the content of the BFS and IBFS steps.

In BFS step (Algorithm 22), for each set $S^{\prime}$ in the current BFS trie and for each input letter $a$ we compute the image $S=S^{\prime} a$ 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 in the (newly formed; 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 the checking subset procedure in line 11, and in consequence, to deal with smaller structures. It also guarantees that $T_{c}$ contains only minimal sets in terms of inclusion (the proof of this fact and all other proofs will be given in the extended version of this paper).

After executing lines 10-15 of Algorithm 2 the trie $T_{v}$ may contains 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 reduction. In our implementation we perform it after the first step and then only when $T_{v}$ contains at least $k$ times more sets since it had after the last reduction (which is the worse case for one step with branching factor $k=|\Sigma|)$.

The IBFS step 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.

One can prove the following
Theorem 1. Given a synchronizing n-state automaton $A=\langle Q, \Sigma, \delta\rangle$, Algorithm 1 returns the shortest length of a reset word for $A$ or reports that no such a word of length $\leq$ maxlen exists.

Proof. 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 $T_{i c}^{j}$ denote $T_{i c}$ after performing $j$ steps of IBFS. Similarly, let $T_{v}^{i}$ denote $T_{v}$ after performing $i$ steps of BFS, and let $T_{v c}^{j}$ denote $T_{i v}$ after performing $j$ steps of IBFS. We have the following

Lemma 1. For each set $S \in T_{c}^{i}$ there is a word $u$ of length $i$, such that $Q u=S$. Similarly for each set $T \in T_{i c}^{j}$ 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 $S^{\prime} \in T_{c}^{i-1}$ (line 5 of Algorithm 2). By induction hypothesis, there is a word $u^{\prime}$ of the length $i-1$ such that $Q u=S^{\prime}$. Hence, $u^{\prime} a$ has length $i$ and we have $Q u^{\prime} a=S^{\prime} a=S$, as required. The proof of the second statement is dual.

```
Algorithm 2 BFS step procedure
    procedure BFS_STEP \(\left(T_{c}, T_{v}\right)\)
        \(L \leftarrow\) EmptyList \(\quad \triangleright\) The list of all new images
        for all \(S^{\prime} \in T_{c}\) do
            for all \(a \in \Sigma\) do
                \(S \leftarrow \delta\left(S^{\prime}, a\right) \quad \triangleright\) Compute the image of \(S^{\prime}\) by the letter \(a\)
                \(L\).insert \((S)\)
            end for
        end for
        \(T_{c} \leftarrow\) EmptyTrie
        for all \(S \in L\) in ascending cardinality order do
            if not \(T_{v}\).Contains_SUBSET_of \((S)\) then
                    \(T_{v} \cdot \operatorname{INSERT}(S)\)
                \(T_{c}\).INSERT \((S)\)
            end if
        end for
        if \(T_{v}\) has grown large since the last reduction then
            \(T_{v}\).REDUCE
        end if
    end procedure
```

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 $T \in T_{i c}^{j}$ such that $S \subseteq T$. By Lemma 1, there are words $u, v$ of lengths $i, j$, respectively, and a state $q \in Q$ such that $Q u=S$ and $\{q\} v^{-1}=T$. Thus, $Q u v=\{q\}$, and $u v$ 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 $T \in T_{i c}^{j}$, and there exists a reset word $w=u x v$ of length $l$, where $|u|=i,|v|=j,|x|=l-i-j$, such that $Q u=S$ 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 $\{q\} \in T_{i c}^{0}$, and a reset word of length $l$ is as required. Assume that the statement is true for all $i^{\prime}+j^{\prime}<$ $i+j$. Assume also, first, that the $(i+j)$-th performed step is BFS one. Then, by the induction assumption there exists a reset word $w^{\prime}=u^{\prime} x^{\prime} v$ of length $l$ and sets $S^{\prime} \in T_{c}^{i-1}$ and $T \in T_{i c}^{j}$ such that $Q u^{\prime}=S^{\prime}$ and $\{q\} v^{-1}=T$ for some state $q \in Q,\left|u^{\prime}\right|=i-1,|v|=j$.

Since $i+j \leq l,\left|x^{\prime}\right|>0$. Let $a$ be the first letter of $x^{\prime}$ and $x^{\prime}=a x^{\prime \prime}$. We need to consider two cases, depending on whether $S^{\prime} a=\delta\left(S^{\prime}, a\right)$ (created in line 5 of Algorithm2) is added (in line 13) into $T_{c}^{i}$ or not. If so, then the statement is true, because we have the reset word $w=w^{\prime}=(u a) x^{\prime \prime} v$ and sets $S=S^{\prime} a \in T_{c}^{i}$ and $T \in T_{i c}^{j}$, with required properties..

Otherwise the reason for not adding $S^{\prime} a$ into $T_{c}^{i}$ must be a set $S \in T_{v}^{i}$, such that $S \subseteq S^{\prime} a$ (line 11). Let $u$ be the word corresponding to $S$ by Lemma 1 Then the word $w=u x^{\prime \prime} v$ (where $x^{\prime}=a x^{\prime \prime}$ ) 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 $|u|+\left|x^{\prime \prime}\right|+|v|<i+(l-(i-1)-j-1)+j=l$, 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, $w=u x^{\prime \prime} v$ is the required word for $i, j$ with $S \in T_{c}^{i}$ and $T \in T_{i c}^{j}, Q u=S$, 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, analogous. The difference is that by the induction assumption, we have now a reset word $w^{\prime}=u x^{\prime} v^{\prime}$, and we take into consideration the last letter of $x^{\prime}$. We leave this part to the reader.

### 2.4 Finding a Reset Word

In order to find a reset word of the found 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 we need to store also the words assigned to these sets. To this end, in line 5 of Algorithm 2 (and analogously in the IBFS procedure) we assign to $S^{\prime}$ 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 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 Heuristics and Optimizations

In addition to the main part of the algorithm described in the previous section we use a number of heuristics and optimizations. They are justified both by experiments and theoretical arguments. Altogether they can reduce computation time by a factor of at least 25 relative to the implementation without these optimizations. We describe briefly only the most important of them.

### 3.1 Estimation of Expected Step Time

To decide which step will be performed in line 12 of the Algorithm $\mathbb{\square}$ we follow the greedy strategy choosing this step whose execution time, together with the goal test, seems to be smaller at the moment. 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. The base for the estimation is the following theoretical result we have established. (A set $S \subset X$ is a random subset of $X$ with Bernoulli distributions in $[q, r]$ if each element $x$ of $X$ is a member of $S$ with probability $p_{x} \in[q, r]$.)

Theorem 2. Let $p, q, r \in(0,1)$ be such that $q \leq r$ and $q>p r$. Let $\mathcal{F}$ be a family of $m$ random subsets of a given set $X$ with Bernoulli distributions in $[q, r]$, and let $S$ be a random subset of $X$ with Bernoulli distributions 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

$$
\left(\frac{1+p}{p}+\frac{1}{q-p r}\right) m^{\log _{w}(1+p)}
$$

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

Proof. Let $f(S, \mathcal{F})$ be the number of visited nodes in the trie constructed for $\mathcal{F}$ by subset checking procedure for $S$.

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

$$
f(S, \mathcal{F})=\sum_{x} g(x, S, \mathcal{F})
$$

where $g(x, S, \mathcal{F})$ is an indicator function taking 1 if the node $x$ is visited and 0 otherwise. By linearity of expectation,

$$
\mathbb{E}[f(S, \mathcal{F})]=\sum_{x} \mathbb{E}[g(x, S, \mathcal{F})]=\sum_{x} \mathbb{P}(g(x, S, \mathcal{F})=1)
$$

We can then group the nodes at the same height:

$$
\sum_{x} \mathbb{P}(g(x, S, \mathcal{F})=1)=\sum_{h=0}^{\infty}\left(\sum_{x \text { at height } h} \mathbb{P}(g(x, S, \mathcal{F})=1)\right)
$$

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 events are independent, since (1) depends only on $S$ and (2) only on $\mathcal{F}$. We may define therefore two indicator functions: $g^{\prime}(x, S)$ which takes the value 1 if the first condition holds (and 0 otherwise) and $g^{\prime \prime}(x, \mathcal{F})$ which takes the value 1 if the second condition holds (and 0 otherwise).

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 the 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 $\min \left\{1, m r^{i}(1-q)^{h-i}\right\}$. Summarizing, for a node $x$ with $i$ ones and $h-i$ zeros on the path we have:

$$
\begin{aligned}
\mathbb{E}\left[g^{\prime}(x, S)\right] & =\mathbb{P}\left[g^{\prime}(x, S)=1\right]=\mathbb{P}(S \text { contains the } i \text { elements specified by } x) \leq p^{i}, \\
\mathbb{E}\left[g^{\prime \prime}(x, \mathcal{F})\right] & =\mathbb{P}\left(g^{\prime \prime}(x, \mathcal{F})=1\right)=\mathbb{P}(\mathcal{F} \text { contains the set encoded by } x) \\
& \leq \min \left\{1, m r^{i}(1-q)^{h-i}\right\}
\end{aligned}
$$

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:

$$
\sum_{x \text { at height } h} \mathbb{P}\left[g^{\prime}(x, S) g^{\prime \prime}(x, \mathcal{F})=1\right] \leq \sum_{i=0}^{h}\binom{h}{i} p^{i} \min \left\{1, m r^{i}(1-q)^{h-i}\right\}
$$

This yields a bound that we will use to estimate

$$
\mathbb{E}[f(S, \mathcal{F})] \leq \sum_{h=0}^{\infty}\left(\sum_{i=0}^{h}\binom{h}{i} p^{i} \min \left\{1, m r^{i}(1-q)^{h-i}\right\}\right)
$$

Let $t=\left\lfloor\log _{w} m\right\rfloor$, where $w=\frac{1+p}{1+p r-q}$. 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 $\sum_{h=0}^{t}\left(\sum_{i=0}^{h}\binom{h}{i} p^{i} \min \left\{1, m r^{i}(1-q)^{h-i}\right\}\right)$. For $\mathbb{P}\left[g^{\prime \prime}(x, \mathcal{F})=1\right]$ we use in this case the trivial bound $\mathbb{P}\left[g^{\prime \prime}(x, \mathcal{F})=1\right] \leq 1$. So, we have the bound

$$
\sum_{h=0}^{t} \sum_{i=0}^{h}\binom{h}{i} p^{i}=\sum_{h=0}^{t}(1+p)^{h}=\frac{(1+p)^{t+1}-1}{p}
$$

Substituting $t=\left\lfloor\log _{w} m\right\rfloor$ yields

$$
\begin{aligned}
\frac{(1+p)^{t+1}-1}{p} & =\frac{(1+p)^{\left\lfloor\log _{w} m\right\rfloor+1}-1}{p} \\
& \leq \frac{(1+p)(1+p)^{\log _{w}(m)}-1}{p} \\
& =\frac{(1+p) m^{\log _{w}(1+p)}-1}{p} \\
& <\frac{(1+p)}{p} m^{\log _{w}(1+p)}
\end{aligned}
$$

Case 2. We estimate $\sum_{h=t+1}^{n}\left(\sum_{i=0}^{h}\binom{h}{i} p^{i} \min \left\{1, m r^{i}(1-q)^{h-i}\right\}\right)$. For this case we use the second bound $\mathbb{P}\left[g^{\prime \prime}(x, \mathcal{F})=1\right] \leq m r^{i}(1-q)^{h-i}$. We obtain

$$
\sum_{i=0}^{h}\binom{h}{i} p^{i} m r^{i}(1-q)^{h-i}=m(1+p r-q)^{h}
$$

and consequently,

$$
\begin{aligned}
\sum_{h=0}^{\infty} m(1+p r-q)^{h+t+1} \leq & \sum_{h=0}^{\infty} m(1+p r-q)^{h+\log _{w} m} \\
= & \sum_{h=0}^{\infty} m\left(m^{\log _{w}(1+p r-q)}(1+p r-q)^{h}\right) \\
= & m^{\log _{w}(w)+\log _{w}(1+p r-q)} \sum_{h=0}^{\infty}(1+p r-q)^{h} \\
& (\text { note that }(1+p r-q)<1, \text { since by assumption } q>p r) \\
< & m^{\log _{w}((1+p r-q) w)} \frac{1}{q-p r} \\
= & \frac{1}{q-p r} m^{\log _{w}(1+p r-q) \frac{1+p}{1+p r-q}} \\
= & \frac{1}{q-p r} m^{\log _{w}(1+p)}
\end{aligned}
$$

Combining both the cases we obtain

$$
\mathbb{E}[f(S, \mathcal{F})]<\frac{1+p}{p} m^{\log _{w}(1+p)}+\frac{m^{\log _{w}(1+p)}}{q-p r}=\left(\frac{1+p}{p}+\frac{1}{q-p r}\right) m^{\log _{w}(1+p)}
$$

as required.
In our empirical observations this optimization reduces computation time by an average of $70 \%$ relative to the implementation performing the BFS and IBFS steps alternatingly. It usually leads to perform slightly more BFS steps, since average sizes of subsets decrease much faster in BFS than increase in IBFS. By a result of Higgins after applying two BFS steps the average size of subsets not greater than $0.55 n$ (see [9]). Our empirical observations show that the two searches meet when the sizes of subsets are as small as $0.03 n$. This fact is also the reason why in the goal test we decided to use subset checking of $T_{c}$ rather than superset checking of $T_{i c}$ (subset checking does not require branching in subtries corresponding to elements not belonging to the queried set).

### 3.2 Adding the IDFS Phase

This is the most important optimization improving not only the performance, but also modifying the general idea. Bidirectional BFS 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 we switch to depth-first search, which has restricted memory requirements, and may use the subsets and words computed so far. Moreover, assuming the Černý conjecture, we may impose an initial limit on the depth of the search, which allows to make the DFS search complete. After each recursive call, when a shorter reset word is found, the limit on the depth of the search is suitably decreased. The search is finished when no limit decreasing is possible and all paths of the limited DFS are exhausted. The search returns either the shortest reset word or a counterexample to the Černý conjecture. The IDFS phase is 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 3.5,

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

### 3.3 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. 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 $s n$, where $s$ is an experimentally established constant, we check if there are unreachable states in $Q$. This is done by the standard DFS search on $Q$. If this is the case, we create a reduced automaton $A^{\prime}$ 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 minimal strongly connected component is on average less than $1-1 / e^{k}$ (provided most automata are synchronizing). From our experiments it follows that for synchronizing automata with $k=2$ this size is $\approx 0.7987 n$. Thus, for example, automata with $n=200$ states are reduced on average by as much as 40 states.

### 3.4 Reordering of the States

Efficiency of operations on radix tries depends on the order in which the input automaton's states are processed. We found that the subset checking is performed faster if the states occurring more frequently in queried subsets are later in the ordering. This is because radix tries tends to have logarithmic height (cf. [6]), and 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. To establish frequencies of occurrences of states, and a preferred initial order based on them, we use a stationary distribution of a Markov chain based on the underlying digraph of the automaton. The details will be given in the extended version of the paper. This optimization is performed before the bidirectional search phase.

The situation changes completely 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. This leads to a different reordering. Both reorderings have been confirmed as optimal by experiments. They show that these optimization reduce computation time by an average of $27 \%$.

### 3.5 Using Heuristic Algorithms and IDFS Shortcut

In order to save a step of search computation we may use known heuristic algorithms to find quickly a good bound for search depth. Therefore, at the beginning of the algorithm, before starting the bidirectional search, we apply a few polynomial time algorithms finding upper bounds for the length of the shortest reset word. In our implementation we use Eppstein algorithm [7, FastSynchro algorithm [13] and our procedure Cut-Off IBFS. The latter is the standard IBFS search with cutting the branches of the search with smallest subsets. This may spare one step in bidirectional search, if the heuristic algorithms find the shortest word.

Yet, more importantly, combined with the IDFS phase, this makes possible to reduce the computation time by several orders of magnitude. 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. For estimation we use the formula in Theorem 2 Our experiments show that this optimization reduces computation time by as much as $89 \%$.

## 4 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 3.1). 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.

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

While the assumptions 2-3 are purely theoretical, they may be treated as an idealization of a typical situation. Using these assumption, denoting by $l$ the length of the shortest reset word of the automaton, we can prove that there exists an integer $0<d<1$, depending on probabilities $p_{c}, p_{i c}$, such that the following holds.

Theorem 3. Under the assumptions (1-5) above, and with l denoting the length of the shortest reset word of the automaton, the expected time complexity of the algorithm is $O\left(k n^{2} r^{l(1+d) / 2)}\right)$, and the space complexity is $O\left(n(k+n)+n r^{l / 2}\right)$.

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

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

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: $\sum_{j=0}^{i} r^{j}=\frac{r^{i+1}-1}{r-1} \in O\left(r^{i}\right)$. Similar bounds hold for $T_{i c}$ and $T_{i v}$, but there are $n$ sets at the beginning, so it yields $O\left(n r^{i}\right)$.

Recall that under assumptions of the theorem we may use Theorem 2, and to obtain the following estimation for the visited number of nodes in the trie

$$
\operatorname{ExpNvN}(m, p, q)=\left(\frac{1+p}{p}+\frac{1}{q-p q}\right) m^{\log _{w}(1+p)}
$$

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

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 $O(n \operatorname{ExpNvN}(m, p, q))$, for suitable $m, p, q$. This is so, because we must count not only visited nodes but also test if the set is a subset of a stored set (which costs $O(n)$, but is done at most once for a visited node).

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

$$
O\left(\left(n r^{i}\right)+\left(k n r^{i}\right)+\left(k n r^{i} \operatorname{ExPNvN}\left(O\left(r^{i}\right), p_{c}, p_{c}\right)\right)\right)
$$

The last component in the sum is dominating, which yields

$$
O\left(k n r^{i}\left(r^{i}\right)^{\log _{w\left(p_{c}, p_{c}\right)}\left(1+p_{c}\right)}\right) .
$$

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

$$
O\left(k n r^{i}\left(n r^{i}\right)^{\log _{w\left(p_{i c}, p_{i c}\right)}\left(1+p_{i c}\right)}\right) .
$$

Considering the goal test, it is enough to count only the goal test time after the IBFS step (multiplied by 2). Considering the goal test, in both cases after the BFS step or IBFS step we can bound the time by

$$
O\left(n r^{i} \operatorname{ExpNvN}\left(O\left(r^{i}\right), p_{i c}, p_{c}\right)\right)=O\left(n^{2} r^{i}\left(r^{i}\right)^{\log _{w\left(p_{i c}, p_{c}\right)}\left(1+p_{i c}\right)}\right)
$$

Computing estimated expected step times after $i$-th BFS step and $i$-th IBFS are done in $O\left(n r^{i}\right)$ (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:

$$
\begin{aligned}
& O\left(k n r^{i}\left(\left(r^{i}\right)^{\log _{w\left(p_{c}, p_{c}\right)}\left(1+p_{c}\right)}+\left(n r^{i}\right)^{\log _{w\left(p_{i}, p_{i c}\right)}\left(1+p_{i c}\right)}\right)+n^{2} r^{i}\left(r^{i}\right)^{\log _{w\left(p_{i c}, p_{c}\right)}\left(1+p_{i c}\right)}\right) \\
\in & O\left(k n^{2} r^{i}\left(r^{i}\right)^{d}\right) \\
= & O\left(k n^{2}\left(r^{i}\right)^{1+d}\right)
\end{aligned}
$$

where

$$
d=\max \left(\left(\log _{w\left(p_{c}, p_{c}\right)}\left(1+p_{c}\right)\right),\left(\log _{w\left(p_{i c}, p_{i c}\right)}\left(1+p_{i c}\right)\right),\left(\log _{w\left(p_{i}, p_{c}\right)}\left(1+p_{i c}\right)\right)\right)
$$

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:

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

The expected space complexity can be bounded by counting stored sets and nodes in the tries after the last step. There are $O\left(r^{l / 2}\right)$ 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 $O\left(n k+n^{2}\right)$ space. So we can state up the space bound for $O\left(n(k+n)+n r^{l / 2}\right)$.

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 $O\left(k n R^{l}\right)$ (assuming we can check visited sets in constant time). Moreover the standard algorithm usually has a larger branching factor $R>r$, since strict supersets of visited sets are not skipped. The expected space complexity also yields an improvement in comparison to the $O\left(n R^{l}\right)$ space bound for the standard BFS.

While, generally, our algorithm is exponential in the length $l$ of the shortest reset word, surprisingly, it works fast in polynomial time for the known series of slowly synchronizing automata, that is those with $l$ close to the Černý bound. These are automata $\mathscr{C}_{n}$ (the Černý automaton), $\mathscr{W}_{n}, \mathscr{D}_{n}^{\prime}, \mathscr{D}_{n}^{\prime \prime}$, and $\mathscr{B}_{n}$ introduced in [1].

Theorem 4. For the class of the Černy automata $\mathscr{C}_{n}$, and the classes a $\mathscr{W}_{n}, \mathscr{D}_{n}^{\prime}, \mathscr{D}_{n}^{\prime \prime}$, and $\mathscr{B}_{n}$ introduced in [1] the algorithm works in $O\left(n^{4}\right)$ time and $O\left(n^{3}\right)$ space.

The proof is based on the exact description of the heuristic mentioned in 3.1 which shows that for each of the mentioned slowly synchronizing automata the algorithm performs mainly IBFS steps (rather than BFS), and the IBFS lists keep containing only one or two sets (due to reductions of visited subsets).

## 5 Experiments

We performed a series of the following experiments for various $n \leq 320$. 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 minimal length of a reset word using the algorithm described in Section 2. On the basis of the obtained results we estimate the expected length of the shortest reset word.

### 5.1 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 2 input letters can be then represented as a sequence of $2 n$ uniformly random natural numbers from $[0, n-1]$. To generate high quality random sequences we have used the WELL number generator [16] (variants 1024 and 19937) seeded by random bytes from /dev/random device. For comparison, recall that Skvortsov and Tipikin, in their experimental study [20, have generated and checked the following numbers of random automata: 2000 automata for each $n \in\{1,2, \ldots, 20,25,30, \ldots, 50\}, 500$ automata for each $n \in\{55,60,65,70\}$, and 200 automata for each $n \in\{75,80, \ldots, 100\}$. In our experiment, up to 7 states, we have computed exact results checking all automata. For each $8 \leq n \leq 100$ we checked one million automata, and for each $101 \leq n \leq 260$ and $n=265,270, \ldots, 320$ we checked 10000 automata. Our computations have been performed mostly on 16 computers with $\operatorname{Intel}(\mathrm{R})$ Core(TM) i7-2600 CPU 3.40 GHz 4 cores and 16 GB of RAM. The algorithm was implemented in $\mathrm{C}++$ and compiled with $\mathrm{g}++$. Distributed computations were managed by a dedicated server and clients applications written in Python.

The average computation time is about 100 or 1000 times faster than the time of Trahtman's program TESTAS [22[23] for automata with 50 states. The reduction to SAT used in [20] 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 0.006 and 0.07 seconds, respectively (we have used faster machines, but only about twice as fast). The Table 1 presents a rough comparison. The average times are relatively small because of rare occurrences of slowly synchronizing automata. We present also the maximum computation time.

Table 1. Comparison of average and maximum computation time for random automata.

| $n$ | 50 | 100 | 150 | 200 | 250 | 300 |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: |
| TESTAS $(\llbracket 22\rfloor)$ | 1.4 s | time-out | - | - | - | - |
| SAT reduction $(\boxed{20})$ | 2.7 s | 70 s | - | - | - | - |
| Our average time | 0.005 s | 0.06 s | 0.469 s | 2.88 s | 31.637 s | 596.249 s |
| Our maximum time | 0.26 s | 3.79 s | 10.12 s | 159.670 s | 5 h 19 min | 7 h 55 min |

### 5.2 Results

Our experiment confirms that for the standard random automata model $\mathbb{A}(n)$ on the binary alphabet the probability that the automaton is synchronizing seems to tend to 1 as the number $n$ of states grows. This conjecture is posed in [20], but we have heard it earlier from Peter Cameron during BCC conference in Exeter 2011. For $n=100$, 2250 of one million automata turned out to be non-synchronizing ( $0.225 \%$ ), and for $n=300$, only five of 10000 automata. The graphical representations of our experiments in this respect forms a smooth curve very fast converging to 1 . We observe also that random automata mostly are not strongly connected.

The main result of our experiments is the estimation of the expected length of the shortest reset word. We deal with the infinite sequence of random variables $\ell(n)$ defined as the length of the shortest reset word for a random synchronizing automaton with $n$ states. We have observed that the approximation $\mathbb{E}[\ell(n)] \approx 1.95 n^{0.55}$ proposed in [20] is inflated. Based on currently available data, we propose a new more precise experimental approximation for the expected length $\mathbb{E}[\ell(n)] \approx 2.5 \sqrt{n-5}$. A comparison of the estimations with the experimentally obtained mean length is given in Figure 1. We observe also that our result suggest that the expected length may belong to $\Theta(\sqrt{n})$.

In contrast with the experiments by Skvortsov and Tipikin [20], our experiments allow also to obtain a good estimation of the approximation error. Making use of the well-known Hoeffding's inequality, we obtain the following:

Theorem 5. Let $M L(n)$ denotes the mean length of the shortest reset word of the automata in the sample of $m$ randomly generated synchronizing $n$-state automata. If the ratio of the automata with the length of the shortest reset word larger than $M_{n}$ to all automata in the sample does not exceed $r$, then with probability at least $1-p$

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

Proof. We make use of the well-known Hoeffding's inequality [10]. Given $0<p \leq 1$, with probability at least $1-p$

$$
\begin{equation*}
|\bar{X}-\mathbb{E}[\bar{X}]| \leq R \sqrt{\frac{\log (2 / p)}{2 m}} \tag{1}
\end{equation*}
$$

where $\bar{X}=\left(X_{1}+\ldots+X_{m}\right) / m$ 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 lengths of the shortest reset words obtained in the experiment are much smaller than the known bounds and that longer lengths occur rarely.

Let $M_{n}$ be the maximal length of the shortest reset word for the $n$-state automata generated in the experiment and $m$ the size of the sample. First we assume that we sample only automata with the length $\leq M_{n}$. Denote the corresponding random variable by $\ell^{\prime}(n)$. Applying the Hoeffding's inequality, putting $X_{1}=\ldots=X_{m}=\ell^{\prime}(n)$, and $R=M_{n}$ we obtain

$$
\left|M L(n)-\mathbb{E}\left[\ell^{\prime}\right]\right| \leq M_{n} \sqrt{\frac{\log (2 / p)}{2 m}}
$$



Fig. 1. Experimental mean length of the shortest reset words compared with estimations.

Let $\ell^{\prime \prime}(n)$ be the length of the shortest reset word for a synchronizing automata with $n$ states $\ell(n) \geq M_{n}$. Then we obtain

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

as required. We have used the well-known bound $n^{3} / 6$ for the length of the shortest reset word.

Assuming the Černý conjecture in the last term $n^{3} / 6$ may be replaced by $(n-1)^{2}$ (giving essentially better estimation). Let us take $n=100, m=10^{6}$ and $p=0.0001$. Since, with probability $q=(1-r)^{m}$ the ratio of the automata with the shortest reset word longer than $M_{n}$ is less than $r$, one may see that for $1 / r \geq 100975, q<0.0001$. Hence, with high probability $1 / r>100975$, and taking into account the experimental value $M_{100}=41$, the error is less than 1.75 (or 0.19 assuming the Cerný 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 $M L(100)=24.34$. Comparing this with the results of Skvortsov and Tipikin [20], we note that, for automata with 100 states, they also have obtained the expected length close to 24 , but the small size of their sample $m=200$ does not allow any reasonable estimation of the error. Other interesting claims of [20] concerning the variance and approximation of $\ell(n)$ will be discussed in the extended version of the paper.

## References

1. D. Ananichev, V. Gusev, and M. Volkov. Slowly synchronizing automata and digraphs. In Mathematical Foundations of Computer Science 2010, volume 6281 of LNCS, pages 55-65. 2010.
2. D. Ananichev and M. Volkov. Synchronizing monotonic automata. In Developments in Language Theory, volume 2710 of $L N C S$, pages 111-121. 2003.
3. Yaakov Benenson, Rivka Adar, Tamar Paz-Elizur, Zvi Livneh, and Ehud Shapiro. DNA molecule provides a computing machine with both data and fuel. Proceedings of the National Academy of Sciences, 100(5):2191-2196, 2003.
4. J. Černý. Poznámka k homogénnym eksperimentom s konečnými automatami. Matematickofyzikálny Časopis Slovenskej Akadémie Vied, 14(3):208-216, 1964. In Slovak.
5. K. Chmiel and A. Roman. COMPAS - A computing package for synchronization. In Implementation and Application of Automata, volume 6482 of $L N C S$, pages 79-86. 2011.
6. L. Devroye. A note on the average depth of tries. Computing, 28:367-371, 1982.
7. D. Eppstein. Reset sequences for monotonic automata. SIAM Journal on Computing, 19:500-510, 1990.
8. M. Gerbush and B. Heeringa. Approximating minimum reset sequences. In Implementation and Application of Automata, volume 6482 of $L N C S$, pages 154-162. 2011.
9. P. Higgins. The range order of a product of i-transformations from a finite full transformation semigroup. Semigroup Forum, 37:31-36, 1988.
10. W. Hoeffding. Probability inequalities for sums of bounded random variables. J. Amer. Statist. Assoc., 58(301):13-30, 1963.
11. Jürgensen, H. Synchronization. Information and Computation, 206(9-10):1033-1044, 2008.
12. J. Kari. Synchronization and stability of finite automata. Journal of Universal Computer Science, 8(2):270-277, 2002.
13. R. Kudłacik, A. Roman, and H. Wagner. Effective synchronizing algorithms. Expert Systems with Applications, 39(14):11746-11757, 2012.
14. D.R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM, 15:514-534, 1968.
15. J. Olschewski and M. Ummels. The complexity of finding reset words in finite automata. In Mathematical Foundations of Computer Science 2010, volume 6281 of LNCS, pages 568-579. 2010.
16. F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. ACM Transactions on Mathematical Software, 32(1):1-16, 2006.
17. A. Roman. New algorithms for finding short reset sequences in synchronizing automata. In International Enformatika Conference (Prague), pages 13-17, 2005.
18. A. Roman. Genetic algorithm for synchronization. In Language and Automata Theory and Applications, volume 5457 of LNCS, pages 684-695. 2009.
19. S. Sandberg. Homing and synchronizing sequences. In Model-Based Testing of Reactive Systems, volume 3472 of $L N C S$, pages 5-33. 2005.
20. E. Skvortsov and E. Tipikin. Experimental study of the shortest reset word of random automata. In Implementation and Application of Automata, volume 6807 of LNCS, pages 290-298. 2011.
21. E. Skvortsov and Y. Zaks. Synchronizing random automata. Discrete Mathematics and Theoretical Computer Science, 12(4):95-108, 2010.
22. A. N. Trahtman. A package TESTAS for checking some kinds of testability. In Implementation and Application of Automata, volume 2608 of $L N C S$, pages 228-232. 2003.
23. A. N. Trahtman. An efficient algorithm finds noticeable trends and examples concerning the Cüný conjecture. In Mathematical Foundations of Computer Science, volume 4162 of LNCS, pages 789800. 2006.
24. A. N. Trahtman. Modifying the upper bound on the length of minimal synchronizing word. In Fundamentals of Computation Theory, volume 6914 of LNCS, pages 173-180. 2011.
25. M. Volkov. Synchronizing automata and the Cerný conjecture. In Language and Automata Theory and Applications, volume 5196 of $L N C S$, pages 11-27. 2008.

[^0]:    * Supported in part by Polish MNiSZW grant N N201 543038.

