An Improved Algorithm for Finding the Shortest Synchronizing Words

Marek Szykuła
Faculty of Mathematics and Computer Science, University of Wrocław, Poland

Adam Zyzik
Faculty of Mathematics and Computer Science, University of Wrocław, Poland

Abstract

A synchronizing word of a deterministic finite complete automaton is a word whose action maps every state to a single one. Finding a shortest or a short synchronizing word is a central computational problem in the theory of synchronizing automata and is applied in other areas such as model-based testing and the theory of codes. Because the problem of finding a shortest synchronizing word is computationally hard, among *exact* algorithms only exponential ones are known. We redesign the previously fastest known exact algorithm based on the bidirectional breadth-first search and improve it with respect to time and space in a practical sense. We develop new algorithmic enhancements and adapt the algorithm to multithreaded and GPU computing. Our experiments show that the new algorithm is multiple times faster than the previously fastest one and its advantage quickly grows with the hardness of the problem instance. Given a modest time limit, we compute the lengths of the shortest synchronizing words for random binary automata up to 570 states, significantly beating the previous record. We refine the experimental estimation of the average reset threshold of these automata. Finally, we develop a general computational package devoted to the problem, where an efficient and practical implementation of our algorithm is included, together with several well-known heuristics.

2012 ACM Subject Classification Theory of computation → Algorithm design techniques; Theory of computation → Formal languages and automata theory

Keywords and phrases Černý conjecture, reset threshold, reset word, subset checking, synchronizing automaton, synchronizing word

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.85

Related Version *Full Version*: `https://arxiv.org/abs/2207.05495`

Supplementary Material *Software (Source Code)*: `https://github.com/marekesz/synchrowords`
archived at `swh:1:dir:2a7194d536e2252dc2a6e18ec5505eb86cb44481`

Funding This work was supported in part by the National Science Centre, Poland under project number 2021/41/B/ST6/03691. The experiments were run on a computational grid of the Institute of Computer Science, University of Wrocław, funded by National Science Centre, Poland, under project number 2019/35/B/ST6/04379.

1 Introduction

A *deterministic finite complete semi-automaton* (called simply an *automaton*) is a 3-tuple (Q, Σ, δ), where Q is a finite set of *states*, Σ is an *input alphabet*, and $\delta\colon Q \times \Sigma \to Q$ is a completely defined *transition function*. The transition function is naturally extended to a function $Q \times \Sigma^* \to Q$. Throughout the paper, by n we denote the number of states in Q and by k we denote the size of the input alphabet Σ. A word is *reset* (or *synchronizing*) if $|\delta(Q, w)| = 1$; in other words, for every two states $p, q \in Q$ we have $\delta(q, w) = \delta(p, w)$. An automaton that admits a reset word is called *synchronizing*.

The classical synchronization problem is, for a given synchronizing automaton, to find a reset word. Preferably, this word should be as short as possible. Therefore, the main property of a synchronizing automaton is its *reset threshold*, which is the length of the shortest reset words. We denote the reset threshold by r. Synchronizing automata and the synchronization problem are known for both their theoretical properties and practical applications.

1.1 Theoretical Developments

On the theoretical side, there is a famous long-standing open problem from 1969 called the Černý conjecture; see an old [44] and a recent survey [18]. The conjecture claims that the reset threshold is at most $(n-1)^2$. If true, the bound would be tight, as the Černý automata meet the bound for each n [8].

Until 2017, the best known upper bound on the reset threshold was $(n^3-n)/6-1 \sim 0.1666\ldots n^3+\mathcal{O}(n^2)$ $(n \geq 4)$ [26] by the well-known Frankl-Pin's bound. The current best known upper bound is $\sim 0.1654n^3+o(n^3)$ by Shitov [37], which was obtained by refining the previous improvement $\sim 0.1664n^3+\mathcal{O}(n^2)$ by Szykuła [39]. Apart from that, better bounds were obtained for many special subclasses of automata. Synchronizing automata are also applied in other theoretical areas, e.g., matrix theory [14], theory of codes [7], Markov processes [43]. Several new results around the topic appear every year. Recently, a special journal issue was dedicated to the problem [45] for the occasion of the 50th anniversary of the problem.

Reset thresholds were also studied for the average case. Berlinkov showed that a random binary automaton is synchronizing with high probability [5]. Moreover, Nicaud showed that such an automaton with high probability has a reset threshold in $\mathcal{O}(n \log^3 n)$ [24]. Based on that, the upper bound $\mathcal{O}(n^{3/2+o(1)})$ on the expected reset threshold of a random binary automaton was obtained [4]. These studies were accompanied by experiments, and the best estimation obtained so far was $2.5\sqrt{n-5}$ [19].

There were also performed massive experiments directly aimed at verifying the Černý conjecture and other theoretical properties for automata with small numbers of states [9, 21, 42]. For all such studies, finding (the length of) a shortest reset word is a crucial problem.

1.2 Synchronization in Applications

Apart from the theory, the synchronization problem finds applications in practical areas, e.g., testing of reactive systems [29, 34], networks [17], robotics [1], and codes [15].

Automata are frequently used to model the behavior of systems, devices, circuits, etc. The idea of synchronization is natural: we aim to restore control over a device whose current state is not known or we do not want our actions to be dependent on it. For instance, for digital circuits, where we need to test the conformance of the system according to its model, each test is an input word and before we run the next one, we need to restart the device. In another setting, we are an observer who knows the structure of the automaton but does not see its current state and wants to eventually learn it by observing the input; once a reset word appears, the state is revealed unambiguously. See a survey [34] explaining synchronizing sequences and their generalization to automata with output: *homing sequences*, which allow determining the (hidden) current state of the automaton by additionally observing the generated automaton's output.

Another particular application comes from the theory of codes, where finite automata act as *decoders* of a compressed input. Synchronizing words can make a code resistant to errors, since if an error occurs, after reading such a word, decoding is restored to the correct path. See a book for the role of synchronization in the theory of codes [7] and recent works [6] explaining synchronization applied to prefix codes.

1.3 Algorithms Finding Reset Words

Determining the reset threshold is computationally hard. The decision problem, whether the reset threshold is smaller than a given integer, is NP-complete [10], and it remains hard even for very restrictive classes such as binary Eulerian automata [47]. The functional problems of computing the reset threshold and a shortest reset word are respectively $\mathrm{FP}^{\mathrm{NP}[\log]}$-complete and $\mathrm{FP}^{\mathrm{NP}}$-complete [25]. Moreover, approximating the reset threshold is hard even for approximation factors in $\mathcal{O}(n^{1-\varepsilon})$, for every $\varepsilon > 0$ [12]. This inapproximatibility also holds for subclasses related to prefix codes [33]. On the other hand, there exists a simple general $\mathcal{O}(n)$-approximating polynomial algorithm [13]. It is open whether there exists a polynomial algorithm approximating within some sublinear factor, e.g., in $\mathcal{O}(n/\log n)$.

From the perspective of fixed-parameter tractability, the main parameter determining the hardness is the reset threshold itself (and the alphabet size, if not fixed). It plays a similar role as the number of variables in the SAT problem, yet, in contrast, it is not given but is the result to be computed. It is trivial to compute the reset threshold in time $\mathcal{O}(n \cdot k^r)$ simply by checking all words of length $1, 2, \ldots, r$ (we need $\mathcal{O}(n)$ time for computing the image of a subset of Q under the action of one letter). This essentially cannot be much better, as assuming the Strong Exponential Time Hypothesis (SETH), the problem cannot be solved in time $\mathcal{O}^*((k-\varepsilon)^r)$, for every $\varepsilon > 0$ [11, Theorem 8], where $\mathcal{O}^*$ suppresses all polynomial factors in the size of the input.

Therefore, exponential exact algorithms that hopefully find a shortest reset word faster in typical or average cases are used. Alternatively, there are many polynomial heuristics proposed that find a relatively short reset word in practice.

1.3.1 Exact Algorithms

In general, exact algorithms can be used for automata that are not too large. They also play an important role in testing heuristics, providing the baseline for comparison (e.g., [30]).

The naive algorithm of checking all words is practically slow, as it does not involve any optimization and works always in time $\mathcal{O}^*(k^r)$. The standard algorithm, e.g., [23, 34, 42], for computing a shortest reset word is finding a path of state subsets in the power automaton (that is, the automaton whose set of states is 2^Q) from Q to a singleton. This works in at most $\mathcal{O}(kn \cdot 2^n)$ time but is practically faster as we usually traverse through fewer sets. Note that n may be much smaller than r (we know examples where r can be quadratic in n, e.g., [2]), but in the average case it is the opposite. The main drawback of this algorithm is its requirement of $\mathcal{O}(2^n)$ space, which is acceptable only up to small $n \sim 30$.

Alternative approaches include utilizing SAT solvers [38] by suitable reductions of the problem and binary search over possible values of r. SAT solvers were also recently tried for partial deterministic finite automata and *careful synchronization* [36]. In the reported results, such solutions reach random binary automata with about 100 states.

The fastest algorithm so far is based on a bidirectional search of the power automaton, equipped with several enhancements [19, 20]. This algorithm was able to deal with binary random automata up to 350 states. Despite several later attempts, no faster solutions were developed and the algorithm was not improved until now.

1.3.2 Heuristic Algorithms

The most classic polynomial algorithm is Eppstein's one [10]. It works in $\mathcal{O}(n^3 + kn^2)$ time and finds a reset word of length at most $(n^3 - n)/3$ (due to the Frankl-Pin's bound). Several heuristic improvements were proposed, e.g., Cycle, SynchroP, and SynchroPL algorithms [23, 42], which do not improve guarantees but behave better in experimental settings, even at the cost of increased worst-case time complexity. Recent works also involve attempts to speed-up heuristics by adapting to parallel and GPU computation [35, 41].

A remarkable heuristic is the *beam* algorithm based on inverse breadth-first search ([30, *CutOff-IBFS*]), i.e., starting from a singleton and ending with Q, which significantly beats other algorithms based on the forward search. Yet, curiously, it does not provide any worst-case guarantees, as it theoretically may not find any reset word at all.

Alternative approaches involve artificial intelligence methods, e.g., hierarchical classifier [28], genetic algorithms [22], and machine learning approaches [27].

1.4 Contribution

We reinvestigate the so-far best exact algorithm [19, 20] and significantly improve it. We develop a series of algorithmic enhancements involving better data structures, decision mechanisms, and reduction procedures. Altogether, we obtain a significant speed-up and decrease the memory requirements. Additionally, the remodeled algorithm is adapted for effective usage of multithreading and GPU computing, which was not possible in the original.

On the implementation side, we develop an open computational package containing the new exact algorithm as well as several known polynomial heuristics. We apply a series of technical optimizations and fine-tune the algorithm to maximize efficiency. The package supports configurable just-in-time compiled computation plans and can be extended with new algorithms.

In the experimental section, we test the efficiency of the algorithm and compute the reset thresholds of binary random automata up to 570 states. We refine the previous estimation formula for the expected reset threshold of these automata.

The computational package is available at [40] (the version related to this paper is 1.1.0).

2 The New Exact Algorithm

Our algorithm is based on the former best exact algorithm [19, 20]. While the new version differs in the choice of data structures and subprocedures, at a high level it is similar and uses two main phases – bidirectional breadth-first search and then inverse depth-first search.

The input to the algorithm is an automaton $\mathscr{A} = (Q, \Sigma, \delta)$ with n states and k input letters. The goal is to find its reset threshold r. In the first step, we check if $\mathscr{A}$ is synchronizing by the well-known procedure [44], which checks for every two states $p, q \in Q$ whether they can be mapped to one state; this is doable in $\mathcal{O}(kn^2)$. Then we get upper bounds on the reset threshold by using polynomial-time heuristics. For this, we use the Eppstein algorithm [10] at first, and then the enhanced *beam* algorithm [30, CutOff-IBFS]. The found upper bound helps the main procedure make better decisions.

Given a subset $S \subseteq Q$ and a word $w \in \Sigma^*$, the *image* of S under the action of w is $\delta(S, w) = \{\delta(q, w) \mid q \in S\}$. The *preimage* of S under the action of w is $\delta^{-1}(S, w) = \{q \in Q \mid \delta(q, w) \in S\}$.

The key idea is to simultaneously run a breadth-first search (BFS) starting from the set Q and computing images, together with an inverse breadth-first search (IBFS) starting from all of the singletons and computing preimages. While both algorithms on their own require computation of at most k^r or at most nk^r sets respectively, combining them lets us compute no more than $nk^{r/2}$ sets, provided that we can somehow test if the searches have met. To do this, we need to check if there exists a pair X, Y of sets, belonging respectively to the BFS and IBFS lists, such that $X \subseteq Y$. Indeed, then we know that there are words $x, y \in \Sigma^*$ such that $X = \delta(Q, x)$ and $Y = \delta^{-1}(\{q\}, y)$ for some $q \in Q$. Because $X \subseteq Y$, we get $\delta(Q, xy) = \{q\}$, which means that xy is a reset word. Due to the *Orthogonal Vectors Conjecture* [16], there is probably no subquadratic solution to this subset problem[1]. Such a solution would also contradict the mentioned fact that we cannot find the reset threshold in $\mathcal{O}^*((k-\varepsilon)^r)$ time assuming SETH. Nevertheless, we employ procedures that work well in our practical case. They are also used to reduce the number of sets in the lists during the searches, which effectively lowers the branching factor. Finally, we do not actually run the two searches until they meet. Instead, we switch to the second phase with inverse DFS (which takes the steps only on the IBFS side computing preimages), when either the memory runs out or we calculate that it should be faster based on the upper bound from the heuristics and the collected statistics.

These high-level ideas are derived from the previous algorithm. However, we design different, more efficient procedures and optimizations for these steps so that it is possible to solve the problem significantly faster and for larger automata. First, we modify data structures and redesign how a single iteration of BFS / IBFS works. Apart from making the bidirectional-search phase faster, it allows completing more iterations before switching to the DFS phase due to the lower memory consumption, which is crucial in the case of large automata. The decision-making part of the algorithm is also extended. We use five types of steps, and the decision on which step to take is based on statistics from the current algorithm's run and forward prediction. In the DFS phase, we enhance the radix trie data structure in terms of both efficiency and memory overhead. We also apply some forms of list reductions, which decrease the branching factor. Finally, every part of the new algorithm can be parallelized in one way or the other, which was not possible before; the general difficulty of parallelization comes from large shared data structures and a lot of branching.

In the next sections, we describe the new algorithmic techniques. For the sake of brevity, we consider a version that only calculates the reset threshold. The algorithm can be trivially modified to also return the reset word by storing pointers to predecessors along with the sets, although then either time or memory footprint is slightly increased.

2.1 Bidirectional Breadth-First Search

The first phase of the algorithm consists of running the two breadth-first searches. The BFS starts with a list L_{BFS} containing just the set Q. When the search starts a new iteration, L_{BFS} is replaced with $\{\delta(S, a) \mid S \in L_{\text{BFS}}, a \in \Sigma\}$. Conversely, L_{IBFS} is initialized with all the singletons and the list is replaced with $\{\delta^{-1}(S, w) \mid S \in L_{\text{IBFS}},\ w \in \Sigma\}$.

We say that the two searches *meet* if there exist $X \in L_{\text{BFS}}$ and $Y \in L_{\text{IBFS}}$ for which $X \subseteq Y$ holds. The meet condition implies that the lists can be reduced by removing the elements which are not minimal (and respectively maximal for IBFS) with respect to inclusion. We can reduce the lists further by ensuring that no new set is a superset (subset for IBFS) of a set belonging to some list from any previous iteration. To make this possible, we keep track

[1] The *Orthogonal Vectors problem* gives two sets A, B of Boolean vectors of the same length and asks if there exists a pair $(u \in A, v \in B)$ such that u and v are orthogonal, i.e., $u \cdot v = \mathbf{0}$. We can reduce our problem to OV by transforming the sets in Y to their complements and then representing all the sets as their characteristic vectors.

of all the visited sets in two additional *history* lists H_{BFS}, H_{IBFS}. This reduction, although usually helpful during most of the iterations, at the end may turn out to be unprofitable, in which case the algorithm will drop the history list(s).

The original idea for the subprocedure to check the meet condition was to keep the lists as dynamic *radix tries*, supporting insertion and subset (or superset) checking operations. Now, instead, we take a somewhat simpler approach and operate directly on the lists, stored as random access containers (such as vectors in C++). We call this subprocedure *MarkSupersets*(A, B) (and a similar one – *MarkProperSupersets*(A, B), which additionally restricts the marked supersets to be non-equal to their subsets).

We split the reductions into three subprocedures: removing duplicates, self-reduction, and then reduction by history. In contrast to performing only one and the most expensive reduction by history (which could also include the first two reductions), after each subprocedure the list size gets smaller, which makes the next one run faster.

Alg. 1 shows the pseudocode of the bidirectional-search phase.

2.1.1 Subset and Superset Checking

There exist several algorithms solving the extremal sets problem in practical settings, e.g., [3]. They take a list of sets and mark all those that are not a subset of any other set.

We use a similar method to those utilizing a lexicographic sort, but we operate on two lists and are allowed to change the order of the second list during computation. *MarkSupersets* (Alg. 2) takes lists A and B and swaps sets in B so that those sets that are supersets of some sets from A appear at the end. The sets are treated like binary strings, i.e., their characteristic vectors, of length n. The procedure recursively splits the sets in A into those containing the d-th state and those not containing it, where d is the recursion depth. In this sense, it works by implicitly building a radix trie on A. We require that A is sorted lexicographically and its elements are unique, which can be guaranteed relatively cheaply before calling the procedure, as sorting is much faster than subset checking. The order gives us the property that the sets containing the d-th state and those not containing it are stored in continuous segments, so we can effectively split A. This lets us simulate in-place trie traversal with a recursive procedure that takes intervals of the lists as inputs. When the intervals are small (determined by the constant parameter *MIN*, Alg. 2 line 2), we can use a brute-force check instead of recursing further, which makes the procedure faster (especially important with GPU).

MarkSupersets is used to reduce the BFS list and to check the meet condition. To implement the *MarkSubsets* procedure needed on the IBFS side, we simply convert the sets to their complements and call *MarkSupersets*. The procedure *MarkProperSupersets* is identical except for checking the containment for a pair of sets, where we additionally check that the two sets are different. When we use multithreading, we split the B list into equal parts after shuffling and execute parallel calls of the procedure. On GPU, we increase the *MIN* parameter and run the brute-force part there.

2.1.2 Decision Procedure

As the algorithm progresses, some steps may become unprofitable. The history lists, though helpful at the beginning, increase memory usage and cause a slow down if used in late iterations. Similarly, list reductions via *MarkSupersets* decrease the branching factor, but they are not that crucial when the search is approaching the reset threshold upper bound.

Algorithm 1 Bidirectional breadth-first search.

Input: A synchronizing automaton $\mathscr{A} = (Q, \Sigma, \delta)$ with $n = |Q|$ states and $k = |\Sigma|$ input letters. An upper bound R on the reset threshold.
Output: Reset threshold r.

```
1:  $L_{\mathrm{BFS}}, H_{\mathrm{BFS}} \leftarrow \{Q\}$
2:  $L_{\mathrm{IBFS}}, H_{\mathrm{IBFS}} \leftarrow \{\{q\} \mid q \in Q\}$
3:  for $r$ from 1 to $R-1$ do
4:      switch CalculateBestStep() do
5:          case BFS
6:              if $H_{\mathrm{BFS}}$ has grown significantly since its last reduction then
7:                  Delete subsets from $H_{\mathrm{BFS}}$ that are larger than the largest ones from $L_{\mathrm{BFS}}$
8:                  Delete non-minimal subsets from $H_{\mathrm{BFS}}$ (MarkProperSupersets)
9:              end if
10:             $L_{\mathrm{BFS}} \leftarrow$ CalculateImages($L_{\mathrm{BFS}}$)
11:             Delete duplicates from $L_{\mathrm{BFS}}$ (lex. sort)
12:             Delete non-minimal subsets from $L_{\mathrm{BFS}}$ (MarkProperSupersets)
13:             Delete supersets of $H_{\mathrm{BFS}}$ from $L_{\mathrm{BFS}}$ (MarkSupersets)
14:             $H_{\mathrm{BFS}} \leftarrow H_{\mathrm{BFS}} \cup L_{\mathrm{BFS}}$
15:         case BFS$^{\mathrm{NH}}$ (without history)
16:             $L_{\mathrm{BFS}} \leftarrow$ CalculateImages($L_{\mathrm{BFS}}$)
17:             Delete duplicates from $L_{\mathrm{BFS}}$ (lex. sort)
18:             Delete non-minimal subsets from $L_{\mathrm{BFS}}$ (MarkProperSupersets)
19:         case IBFS
20:             ...                                              ▷ Analogous to BFS
21:         case IBFS$^{\mathrm{NH}}$ (without history)
22:             ...                              ▷ Analogous to BFS (without history)
23:         case DFS
24:             DFS(BuildStaticTrie($L_{\mathrm{BFS}}$), $L_{\mathrm{IBFS}}$, $r$, $R$)
25:             return $R$                       ▷ DFS sets $R \leftarrow$ the reset threshold
26:     if MarkSupersets($L_{\mathrm{BFS}}, L_{\mathrm{IBFS}}$) has found at least one superset then
27:         return $r$
28:     end if
29: end for
30: return $R$
```

We distinguish five types of steps from which the algorithm always chooses one for the next iteration – DFS, BFS, IBFS, BFS without the history list (denoted by BFS$^{\mathrm{NH}}$) and IBFS without the history list (denoted by IBFS$^{\mathrm{NH}}$). To assess which option to choose, we roughly estimate the cost subset checking operations each of them will require.

We reuse some of the equations previously defined in [20]. In particular, under simplifying assumptions about the uniform distribution of the states in sets we take their upper bound from [20, Theorem 4] previously applied to tries. Since our procedure can be interpreted as building a trie implicitly on the fly, this bound can also serve as a rough bound on the expected number of subset checking operations in a call to *MarkSupersets*. Let A_s, B_s be the size of the lists and A_d, B_d be their densities, i.e., for a list L let $\mathit{density}(L) = \frac{\sum_{S \in L} |S|}{n|L|}$.

Algorithm 2 Recursive procedure *MarkSupersets*.

```
Input: Lexicographically sorted list intervals $A$ and $B$ of sets with unique elements. Current
    depth of recursion $d$ ($d = 0$ for the initial call).
 1: procedure MarkSupersets($A$, $B$, $d$)
 2:     if $|A| < MIN$ then                                      ▷ Brute-force for small $|A|$
 3:         Check each pair in $A \times B$ and delete the supersets from $B$ (move at the end and
    shrink the interval).
 4:     else
 5:         $A_0, A_1 \leftarrow A$ split by the $d$-th bit       ▷ $A$ is sorted, so a binary search suffices
 6:         $MarkSupersets(A_0, B, d+1)$
 7:         Sort $B$ by the $d$-th bit.                          ▷ Linear time scan
 8:         $B_1 \leftarrow$ interval in which the $d$-th bit is set        ▷ Suffix of $B$
 9:         $MarkSupersets(A_1, B_1, d+1)$
10:     end if
11: end procedure
```

Then, analogously to [20, ExpNvn in 4.2], we define:

$$\mathit{ExpMark}(A_s, B_s, A_d, B_d) = B_s\left(\frac{1+B_d}{B_d} + \frac{1}{A_d - A_d B_d}\right) A_s^{\log_w(1+B_d)},$$

where $w = (1+B_d)/(1+A_d B_d - A_d)$.

To estimate the sizes of the lists after (and in between) the reductions, we store the ratios $r^{\mathit{dupl}}_{\mathrm{BFS}}, r^{\mathit{self}}_{\mathrm{BFS}}, r^{\mathit{hist}}_{\mathrm{BFS}}$ of the reduced sets respectively by the removal of duplicates, the removal of non-minimal subsets, and the reduction by history list. For instance, $r^{\mathit{dupl}}_{\mathrm{BFS}}$ is the fraction of the removed duplicates during the first reduction. Similarly, separate ones are stored for the IBFS counterpart. In addition to the cost of subset checking, we also add the cost of computing sets themselves and reduction of duplicates (the constant *SetCost*, set to 512 in the implementation); however, in most cases, the cost of subset checking is dominant.

For instance, the cost of the next BFS step is calculated as follows:

$$\begin{aligned}
\mathit{BFS}_{\mathit{cost}} = {}& \mathit{SetCost} \cdot k \cdot |L_{\mathrm{BFS}}| && (1)\\
&+ \mathit{ExpMark}(k \cdot (1 - r^{\mathit{dupl}}_{\mathrm{BFS}}) \cdot |L_{\mathrm{BFS}}|, \quad k \cdot (1 - r^{\mathit{dupl}}_{\mathrm{BFS}}) \cdot |L_{\mathrm{BFS}}|, && (2)\\
&\qquad \mathit{density}(L_{\mathrm{BFS}}), \quad \mathit{density}(L_{\mathrm{BFS}}))\\
&+ \mathit{ExpMark}(|H_{\mathrm{BFS}}|, \quad k \cdot (1 - r^{\mathit{dupl}}_{\mathrm{BFS}}) \cdot (1 - r^{\mathit{self}}_{\mathrm{BFS}}) \cdot |L_{\mathrm{BFS}}|, && (3)\\
&\qquad \mathit{density}(H_{\mathrm{BFS}}), \quad \mathit{density}(L_{\mathrm{BFS}}))\\
&+ \mathit{ExpMark}(k \cdot (1 - r^{\mathit{dupl}}_{\mathrm{BFS}}) \cdot (1 - r^{\mathit{self}}_{\mathrm{BFS}}) \cdot (1 - r^{\mathit{hist}}_{\mathrm{BFS}}) \cdot |L_{\mathrm{BFS}}|, \quad |L_{\mathrm{IBFS}}|, && (4)\\
&\qquad \mathit{density}(L_{\mathrm{BFS}}), \quad \mathit{density}(L_{\mathrm{IBFS}})).
\end{aligned}$$

The formula is the sum of the four costs: the set cost (1), which estimates the cost of computing the successors' list with images and reduction of duplicates, the self-reduction cost (2), which assumes that the list size was already reduced by the factor $r^{\mathit{dupl}}_{\mathrm{BFS}}$, the reduction by history cost (3), which assumes both preceding reductions, and the meet condition check cost (4).

The formulas for $\mathrm{BFS}^{\mathrm{NH}}$, IBFS, and $\mathrm{IBFS}^{\mathrm{NH}}$ are analogous. Additionally, if we cannot perform a step because there is not enough memory, we set its expected cost to ∞. Once we choose $\mathrm{BFS}^{\mathrm{NH}}$, we free the history and no longer consider the BFS step with it, so in this case, we also set its cost to ∞ (this is symmetrical for $\mathrm{IBFS}^{\mathrm{NH}}$ and IBFS).

Next, we try to predict the full costs of choosing these steps by estimating the number of operations under the assumption that the algorithm will transition into the DFS phase one iteration later (or in the current iteration in the case of the *DFS* option). First, we calculate the expected branching factor in the DFS phase

$$f = k \cdot (1 - r_{\mathrm{IBFS}}^{dupl} \cdot \mathit{DFSReductionOfReduction}),$$

where we take the average reduction of duplicates from the IBFS step and reduce it by some factor for a more pessimistic estimation ($\mathit{DFSReductionOfReduction} = 1/k$ in the implementation). r_{IBFS}^{dupl} is the ratio of duplicates removed in the IBFS list during the latest reduction (since the DFS also removes duplicates). We assume conservatively that the reset threshold is equal to the known upper bound R and from that, we get the estimated number of iterations $R - r$ that still need to be done, where r is the number of the current iteration, which just begins. The predicted full cost of the BFS option is as follows (the number of sets multiplied by the set computing costs together with the meet condition cost):

$$\begin{aligned} \mathit{DFS}_{pred} = {} & f \cdot \frac{f^{R-r} - 1}{f - 1} \cdot (\mathit{SetCost} \cdot \mathit{DFSSetCostWeight} \cdot k/f \\ & + \mathit{DFSCheckCostWeight} \cdot \mathit{ExpMark}(|L_{\mathrm{BFS}}|, |L_{\mathrm{IBFS}}|, \mathit{density}(L_{\mathrm{BFS}}), \mathit{density}(L_{\mathrm{IBFS}}))\ ), \end{aligned}$$

where *DFSSetCostWeight* and *DFSCheckCostWeight* are constants (both set to 0.25 in the implementation) compensating for the fact that the operations are faster in the DFS phase, as additional optimizations are possible.

Finally, the step with the lowest predicted full cost is chosen. As an exception to this rule, if we do not expect to switch into the DFS phase soon (the steps without the history have both larger costs than the regular steps), instead of comparing the prediction costs of every choice, we consider only the BFS and IBFS costs and greedily choose the one with the lower single-step cost. This makes the bidirectional search more balanced in the short term. Otherwise, BFS is strongly preferred due to the assumption that the rest of the steps will be (inverse) DFS, which computes preimages, but the early statistics are less relevant for its estimation.

2.1.3 Heuristic Upper Bound

Having a good upper bound R on the reset threshold, preferably tight, is crucial for making the right decisions, i.e., not giving away history or entering the DFS phase too late. On the other hand, we do not want to spend too much time computing the bound.

The beam algorithm has its parameter *beam size*, which limits the size of the list and thus directly controls the quality (i.e., how close R is to the reset threshold) and complexity trade-off. Instead of setting the beam size to be a function of n ([20, 30]), we use an adaptive approach. We run it first with a relatively small beam size to find some reasonable bound, and then use this bound to calculate a rough estimation of the running time of the exact algorithm. The beam size is selected so that the beam algorithm's cost is a small fraction of that of the exact algorithm. In practice, this means that the beam size is larger for automata with larger (upper bounds on) reset thresholds than for those automata with smaller ones, even when n is the same.

2.2 Depth-First Search

In the second phase, the algorithm switches to an inverse depth-first search, which allows staying within the memory limit by adjusting the maximum list size. During this phase, the steps are taken only on the IBFS side. The fact that the BFS list no longer changes allows the meet condition check to be optimized.

2.2.1 Static Radix Trie

The L_{BFS} list is stored in an optimized data structure that supports the *ContainsSubset* operation. It is based on a radix trie, in which the characteristic vectors of the sets are stored. The queries rely on traversing the trie and, in each step, descending to either both children or just the left (zero) child, depending on the queried set.

In contrast with the usual radix trie, we apply a few specific optimizations:

Variable state ordering. In each node at a depth d, instead of splitting the subtrees by the d-th state, we split by a state x chosen specifically for the node. The state x is chosen so that the number of sets stored in the current subtree that also contain x is the largest possible. In practice, this makes the queries faster, since, in vertices such that x does not belong to the queried set, a large number of sets is immediately skipped.
This optimization also implies path compression, as we do not have nodes that do not split the sets into two non-empty parts. To ensure this, we also exclude the case that x is contained in all the sets and do not use it as a division state.

Leaf threshold. For a fixed constant parameter *MIN* ($= 10$ in the implementation), when the number of sets in a subtree is less or equal to *MIN*, we store them all in one vertex and do not recurse further. This does not increase running time and lowers the memory overhead. Technically, we can already store the sets in a node whose left (zero) subtree contains at most *MIN* sets, which avoids creating an additional node.

Joint queries. Instead of checking each subset separately, we group them by the same cardinality and check one group in a call using the swapping technique as in Alg. 2. Grouping saves operations responsible for traversing the trie, and additionally, grouping by cardinality allows to make cheap size elimination checks as below.

Size elimination. Every node v stores the minimal size m_v of the sets in its subtree. When we query for subsets of size s, if $m_v \leq s$ does not hold, we do not recurse into the subtree of v. This enhancement is derived from the previous algorithm [20], yet due to joint queries, we make these checks cheaper by executing at most one such check in a node for one cardinality. A similar optimization from the previous algorithm is mask elimination – checking if the intersection of all subsets in the subtree is contained in the queries set, yet we do not use it as it does not improve performance in our case.
Our trie can be built in time $\mathcal{O}(|L_{\mathrm{BFS}}| \cdot n^2)$ ($< n$ levels processed in time $\mathcal{O}(|L_{\mathrm{BFS}}| \cdot n)$).

2.2.2 DFS Procedure

During the search, at each depth, the current list is split into parts of size at most $\mathit{available\_memory}/\big((k+1)(R-r)\big)$, where $R-r$ is the upper bound on the remaining number of steps to be done. Then, it recurses with each of these parts one by one, to make sure we do not run out of memory. The elements are sorted in order of descending cardinality so that the most promising sets are recursed first, which in turn can quickly improve the upper bound if it was not tight. The cardinality sort is also necessary for joint queries. The lists are reduced by the removal of duplicates and calls to *MarkSubsets* only once every few iterations, which still lowers the branching factor significantly. Parallel computation is performed by a thread pool with tasks being these separate calls for each cardinality.

3 Experiments

The implementation used for experiments is available at [40]. The tests were using `exact_reduce` configuration. The *old* algorithm [20] was run with the original code provided to us by the authors. The experiments with time measurement were run on computers with

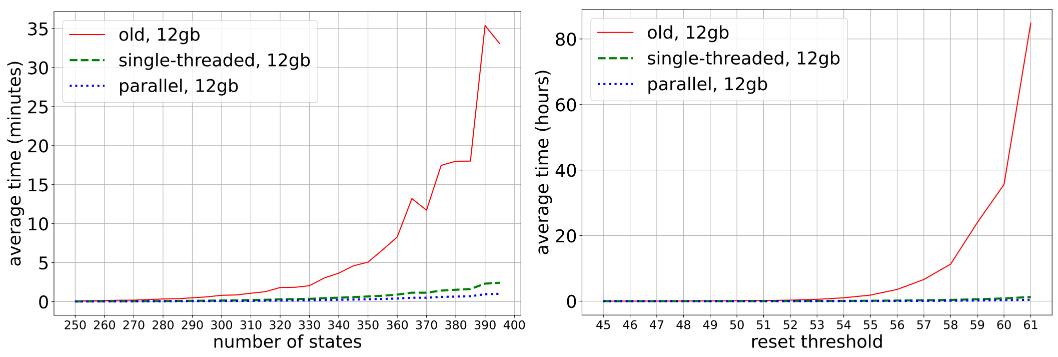

Figure 1 The mean running time for random binary automata with different numbers of states (left) and with different reset thresholds (right).

AMD Ryzen Threadripper 3960X 24-Core Processor, 64GB RAM, and two RTX3080 Nvidia GPU cards. We compiled the code using `gcc 9.3.0` and `nvcc 10.1` (run with `gcc 7.5.0`). We have tested the algorithm in both the *single-threaded* (only one thread without GPU) and *parallel* modes (6 threads and GPU enabled).

A *random* automaton with n states and k letters is generated by choosing each transition $\delta(q, a) \in Q$ uniformly at random, for $q \in Q$, $a \in \Sigma$. There is a negligible number of non-synchronizing automata obtained in this way, which were excluded (cf. [5]).

3.1 The Efficiency

Fig. 1 shows the efficiency comparison of our algorithm in both single-threaded and parallel modes, together with the old algorithm. This experiment was run for 1,000 random binary automata for each $n \in \{250, 255, \dots, 395\}$. We managed to test automata with up to 395 with the old algorithm, which took 3,177h computation time of a single process in total (we have used more memory, better hardware, and computed 10 times fewer automata per n than in the original experiments [20], which were done up to $n = 350$). The (unfinished) attempt to compute $1,000$ automata with $n = 400$ by the old algorithm took over 770h, whereas our algorithm (single-threaded) finished in 48h. The total computation time of our algorithm up to $n = 395$ was resp. 299h in the single-threaded mode and 135h in the parallel mode.

Fig. 2 shows the average increase in running time when the reset threshold grows. The general observed tendency is a factor of about 1.5 for our algorithm.

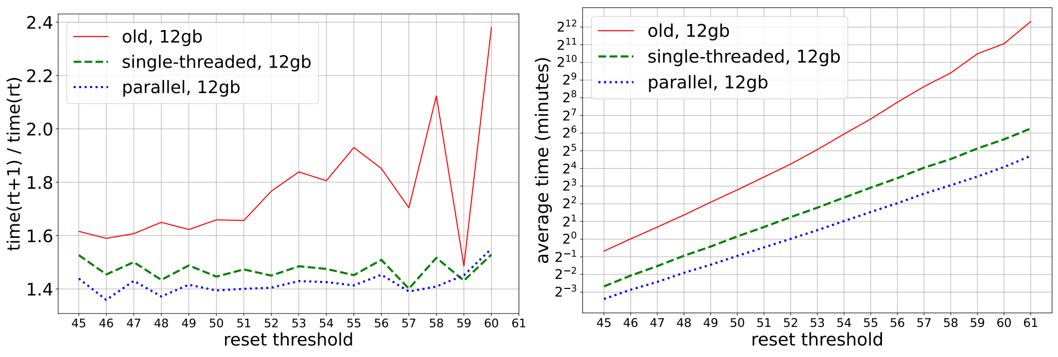

Figure 2 The mean running time growth factor in relation to reset threshold (left) and the mean running time in a logarithmic scale (right).

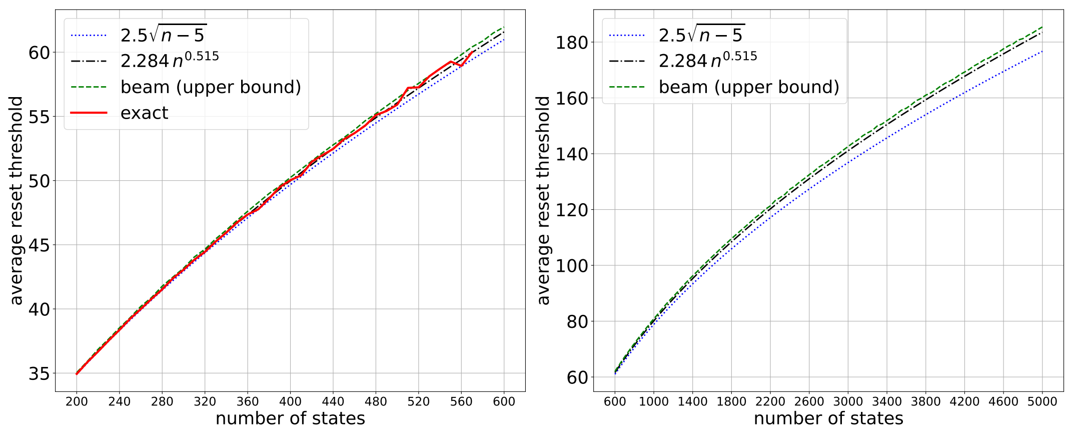

Figure 3 The mean reset threshold of binary automata with n states. For every $n \in \{5, 10, \dots, 300\}$, $n \in \{305, 310, \dots, 400, 410, 420, \dots, 500\}$, and $n \in \{510, 520, \dots, 570\}$ we calculated resp. 10,000, 1,000, and 100 automata.

Hard instances

Most automata have their reset thresholds sublinear, but there exist other examples (though they are rare). As the reset threshold is the main indicator of difficulty, even instances with a small number of states should be difficult for algorithms.

From the known constructions, the most extreme automata with respect to the reset threshold are *slowly synchronizing* ones [2]. They have reset thresholds close to $(n-1)^2$, and the Černý series meets this bound. Yet, they all have the property that the IBFS list reduces to a constant number of sets in every iteration, thus our algorithm works in polynomial time for them, just as the old algorithm. An example of extreme automata without this property could be the unique series of automata with a *sink* state (i.e., a state $q \in Q$ such that $\delta(q, a) = q$ for all $a \in \Sigma$), that reaches the maximum reset threshold $n(n-1)/2$ in this class [31]. (A synchronizing automaton can have at most one sink state and a reset word must map all the states to it.) A *slowly sink* automaton from this series with 26 states has 25 letters and its reset threshold 325 is computed by the old algorithm in 34m 14s, whereas the new algorithm computes it in 7m 22s (parallel).

3.2 Mean reset threshold

In the second experiment, we computed reset thresholds of random binary automata with $n \in \{410, 420, \dots, 570\}$ states with our algorithm in parallel mode. The mean computation time for $n = 500$ was 9m 42s. Fig. 3 shows the mean reset threshold. In addition, up to $n = 5{,}000$ we computed a good upper bound using the beam algorithm with beam size $n \log n$ (we made the beam algorithm much faster due to GPU computation; the previous such experiments were done up to $n = 1000$ [30]).

It is now visible that the previous formula $2.5\sqrt{n-5}$ is underestimated. On the other hand, a standard approach[2] of deriving a function of the form $a(n+b)^c + d$ yields a wrong formula, lately exceeding the upper bound obtained by the beam. We decrease the exponent to fit with a more accurate estimation $2.284\,n^{0.515}$.

[2] We use the algorithm from `scipy.optimize.curve_fit` with the Levenberg-Marquardt algorithm.

4 Conclusions

We have improved the best-known algorithm for computing the (length of the) shortest reset words. While the overall idea of employing bidirectional breadth-first search is the same, we replace each of its subprocedures with more efficient ones.

The algorithm can be easily adapted to alternative synchronization settings and other related problems. For instance, it is trivial to use it for *careful* synchronization [36], where some transitions can be forbidden. It can also be adapted to, e.g, non-careful settings [6], *mortal words* [32], or subset synchronization [46].

For future work, we plan to use this new algorithmic tool to perform more extensive experiments concerning reset thresholds, especially with larger automata and with a larger alphabet. Finally, it can be used to experimentally verify or extend the current verification range of certain conjectures.

References

1 D. S. Ananichev and M. V. Volkov. Synchronizing monotonic automata. In *Developments in Language Theory*, volume 2710 of *LNCS*, pages 111–121. Springer, 2003.

2 D. S. Ananichev, M. V. Volkov, and V. V. Gusev. Primitive digraphs with large exponents and slowly synchronizing automata. *Journal of Mathematical Sciences*, 192(3):263–278, 2013.

3 R. J. Bayardo and B. Panda. *Fast Algorithms for Finding Extremal Sets*, pages 25–34. SIAM, 2011.

4 M. Berlinkov and M. Szykuła. Algebraic synchronization criterion and computing reset words. *Information Sciences*, 369:718–730, 2016.

5 M. V. Berlinkov. On the Probability of Being Synchronizable. In *Proceedings of the Second International Conference on Algorithms and Discrete Applied Mathematics - Volume 9602*, volume 9602 of *CALDAM*, pages 73–84. Springer, 2016.

6 M. V. Berlinkov, R. Ferens, A. Ryzhikov, and M. Szykuła. Synchronizing Strongly Connected Partial DFAs. In *STACS*, volume 187 of *LIPIcs*, pages 12:1–12:16. Schloss Dagstuhl, 2021.

7 J. Berstel, D. Perrin, and C. Reutenauer. *Codes and Automata*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2009.

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

9 M. de Bondt, H Don, and H. Zantema. Lower Bounds for Synchronizing Word Lengths in Partial Automata. *Int. J. Found. Comput. Sci.*, 30(1):29–60, 2019.

10 D. Eppstein. Reset sequences for monotonic automata. *SIAM Journal on Computing*, 19:500–510, 1990.

11 H. Fernau, P. Heggernes, and Y. Villanger. A multi-parameter analysis of hard problems on deterministic finite automata. *Journal of Computer and System Sciences*, 81(4):747–765, 2015.

12 P. Gawrychowski and D. Straszak. Strong inapproximability of the shortest reset word. In *Mathematical Foundations of Computer Science*, volume 9234 of *LNCS*, pages 243–255. Springer, 2015.

13 M. Gerbush and B. Heeringa. Approximating minimum reset sequences. In *Implementation and Application of Automata*, volume 6482 of *LNCS*, pages 154–162. Springer, 2011.

14 B. Gerencsér, V. V. Gusev, and R. M. Jungers. Primitive Sets of Nonnegative Matrices and Synchronizing Automata. *SIAM J. Matrix Anal. Appl.*, 39(1):83–98, 2018.

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

16 D. M Kane and R. R. Williams. The Orthogonal Vectors Conjecture for Branching Programs and Formulas. In Avrim Blum, editor, *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*, volume 124 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

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

18 J. Kari and M. V. Volkov. Černý conjecture and the road colouring problem. In *Handbook of automata*, volume 1, pages 525–565. European Mathematical Society Publishing House, 2021.

19 A. Kisielewicz, J. Kowalski, and M. Szykuła. A Fast Algorithm Finding the Shortest Reset Words. In *COCOON*, volume 7936 of *LNCS*, pages 182–196, 2013.

20 A. Kisielewicz, J. Kowalski, and M. Szykuła. Computing the shortest reset words of synchronizing automata. *Journal of Combinatorial Optimization*, 29(1):88–124, 2015.

21 A. Kisielewicz, J. Kowalski, and M. Szykuła. Experiments with Synchronizing Automata. In *Implementation and Application of Automata*, volume 9705 of *LNCS*, pages 176–188. Springer, 2016.

22 J. Kowalski and A. Roman. A new evolutionary algorithm for synchronization. In Giovanni Squillero and Kevin Sim, editors, *Applications of Evolutionary Computation*, pages 620–635. Springer, 2017.

23 R. Kudłacik, A. Roman, and H. Wagner. Effective synchronizing algorithms. *Expert Systems with Applications*, 39(14):11746–11757, 2012.

24 C. Nicaud. Fast Synchronization of Random Automata. In Klaus Jansen, Claire Mathieu, José D. P. Rolim, and Chris Umans, editors, *APPROX/RANDOM 2016*, volume 60 of *LIPIcs*, pages 43:1–43:12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

25 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. Springer, 2010.

26 J.-E. Pin. On two combinatorial problems arising from automata theory. In *Proceedings of the International Colloquium on Graph Theory and Combinatorics*, volume 75 of *North-Holland Mathematics Studies*, pages 535–548, 1983.

27 I. Podolak, A. Roman, M. Szykuła, and B. Zieliński. A machine learning approach to synchronization of automata. *Expert Systems with Applications*, 97:357–371, 2018.

28 I. T. Podolak, A. Roman, and D. Jędrzejczyk. Application of hierarchical classifier to minimal synchronizing word problem. In *Artificial Intelligence and Soft Computing*, volume 7267 of *LNCS*, pages 421–429. Springer, 2012.

29 I. Pomeranz and S.M. Reddy. On achieving complete testability of synchronous sequential circuits with synchronizing sequences. *IEEE Proc. International Test Conference*, pages 1007–1016, 1994.

30 A. Roman and M. Szykuła. Forward and backward synchronizing algorithms. *Expert Systems with Applications*, 42(24):9512–9527, 2015.

31 I. K. Rystsov. Reset words for commutative and solvable automata. *Theoretical Computer Science*, 172(1-2):273–279, 1997.

32 A. Ryzhikov. Mortality and Synchronization of Unambiguous Finite Automata. In Robert Mercaş and Daniel Reidenbach, editors, *Combinatorics on Words*, pages 299–311. Springer International Publishing, 2019.

33 A. Ryzhikov and M. Szykuła. Finding Short Synchronizing Words for Prefix Codes. In *MFCS 2018*, volume 117 of *LIPIcs*, pages 21:1–21:14. Schloss Dagstuhl, 2018.

34 S. Sandberg. Homing and synchronizing sequences. In *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 5–33. Springer, 2005.

35 N. E. Saraç, O. F. Altun, K. T. Atam, S. Karahoda, K. Kaya, and H. Yenigün. Boosting expensive synchronizing heuristics. *Expert Systems with Applications*, 167:114203, 2021.

36 H. Shabana. Exact synchronization in partial deterministic automata. *Journal of Physics: Conference Series*, 1352:012047, 2019.

37 Y. Shitov. An Improvement to a Recent Upper Bound for Synchronizing Words of Finite Automata. *Journal of Automata, Languages and Combinatorics*, 24(2–4):367–373, 2019.

38 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. Springer, 2011.

39 M. Szykuła. Improving the Upper Bound on the Length of the Shortest Reset Word. In *STACS 2018*, LIPIcs, pages 56:1–56:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

40 M. Szykuła and A. Zyzik. Synchrowords. `https://github.com/marekesz/synchrowords`, 2022.

41 M. K. Taş, K. Kaya, and H. Yenigün. Synchronizing billion-scale automata. *Information Sciences*, 574:162–175, 2021.

42 A. N. Trahtman. An efficient algorithm finds noticeable trends and examples concerning the Černý conjecture. In *Mathematical Foundations of Computer Science*, volume 4162 of *LNCS*, pages 789–800. Springer, 2006.

43 N. F. Travers and J. P. Crutchfield. Exact Synchronization for Finite-State Sources. *Journal of Statistical Physics*, 145(5):1181–1201, 2011.

44 M. Volkov. Synchronizing automata and the Černý conjecture. In *Language and Automata Theory and Applications*, volume 5196 of *LNCS*, pages 11–27. Springer, 2008.

45 M. V. Volkov, editor. *Special Issue: Essays on the Černý Conjecture*, volume 24 (2–4) of *Journal of Automata, Languages and Combinatorics*, 2019.

46 V. Vorel. Subset Synchronization and Careful Synchronization of Binary Finite Automata. *International Journal of Foundations of Computer Science*, 27(05):557–577, 2016.

47 V. Vorel. Complexity of a problem concerning reset words for Eulerian binary automata. *Information and Computation*, 253:497–509, 2017.