Completely Reachable Automata: A Polynomial Algorithm and Quadratic Upper Bounds

Robert Ferens
University of Wrocław, Poland

Marek Szykuła
University of Wrocław, Poland

Abstract

A complete deterministic finite (semi)automaton (DFA) with a set of states Q is *completely reachable* if every non-empty subset of Q can be obtained as the image of the action of some word applied to Q. The concept of completely reachable automata appeared several times, in particular, in connection with synchronizing automata; the class contains the Černý automata and covers a few separately investigated subclasses. The notion was introduced by Bondar and Volkov (2016), who also raised the question about the complexity of deciding if an automaton is completely reachable. We develop a polynomial-time algorithm for this problem, which is based on a new complement-intersecting technique for finding an extending word for a subset of states. The algorithm works in $\mathcal{O}(|\Sigma| \cdot n^3)$ time, where $n = |Q|$ is the number of states and $|\Sigma|$ is the size of the input alphabet. Finally, we prove a weak Don's conjecture for this class of automata: a subset of size k is reachable with a word of length smaller than $2n(n-k)$. This implies a quadratic upper bound in n on the length of the shortest synchronizing words (reset threshold) for the class of completely reachable automata and generalizes earlier upper bounds derived for its subclasses.

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

Keywords and phrases Černý conjecture, complete reachability, DFA, extending word, reachability, reset threshold, reset word, simple idempotent, synchronizing automaton, synchronizing word

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.59

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2208.05956`

Funding *Robert Ferens*: Supported in part by the National Science Centre, Poland under project number 2017/26/E/ST6/00191.
Marek Szykuła: Supported in part by the National Science Centre, Poland under project number 2021/41/B/ST6/03691.

Acknowledgements We thank the anonymous reviewer for careful proofreading and his suggestions.

1 Introduction

The concept of completely reachable automata origins from the theory of synchronizing automata. A deterministic finite automaton is *synchronizing* if starting from the set of all states, after reading a suitable *reset* word, we can narrow (reach) the set of possible states to a singleton. On the other hand, an automaton is *completely reachable* if starting from the set of all states, we can reach every non-empty subset of states. Thus, every completely reachable automaton is synchronizing, and the class of completely reachable automata forms a remarkable subclass of (synchronizing) automata.

Synchronizing automata are most famous due to a longstanding open problem: the Černý conjecture, which states that every synchronizing n-state automaton admits a reset word of length at most $(n-1)^2$. Applications of synchronizing automata include testing of reactive systems [20] and synchronization of codes [2, 4]. The currently best upper bound for the Černý problem is cubic in n [18, 21, 22]. Finding better bounds for particular cases was a topic of extensive study. Some of the related computational problems such as checking if an automaton is synchronizing are solved in polynomial time [11]. Other ones, such as finding a reset word of the smallest length, are hard, but practical (heuristic and optimizing) methods are developed [23]. Most of the research on the topic of synchronizing automata was collected and comprehensively described in the recent survey [25]; we also refer to older ones [16, 24].

Other studies where completely reachable automata appear include the descriptional complexity of formal languages. Here, the complete reachability of an automaton is often juxtaposed with the maximality of the state complexity or the syntactic complexity of languages recognized by automata [5, 15, 17].

The notion of completely reachable automata was first introduced in 2016 by Bondar and Volkov [6], who also asked about the complexity of the computational problem of deciding whether a given automaton is completely reachable. The analogous decision problem for synchronizing automata is easily solvable in quadratic time in the number of states. Later studies revealed a connection to the so-called Rystsov graph, whose generalization can be used to characterize completely reachable automata [5, 7]. However, this does not yet lead to an effective algorithm, as it is unknown how to compute these graphs. Recently, the case of binary automata was solved with a quasilinear-time algorithm [8], which strongly relies on the specificity of both letters when they ensure the complete reachability of the automaton.

The class of completely reachable automata contains several previously studied cases. It contains the Černý automata [9], which meet the conjectured bound $(n-1)^2$ for the Černý problem, and some of the so-called *slowly synchronizing series* [1]. The class of automata with the full transition monoid [13], synchronizing automata with simple idempotents [19] and aperiodically 1-contracting automata [10] are proper subclasses of completely reachable automata. For the first two subclasses and a special case of the third one quadratic upper bounds on the length of the shortest reset words instead of a cubic one were established for the Černý problem. However, for the whole class of completely reachable automata, only cubic bounds were known (though better than in the general case) [5].

In connection to the bounds, a remarkable conjecture and a generalization of the Černý one is Don's conjecture [10, Conjecture 18]. It states that for an n-state automaton, for every $1 \leq k < n$, if a subset of states of size k is reachable, then it can be reached with a word of length at most $n(n-k)$. This conjecture was disproved in general [14] but weaker versions were proposed: one restricting to completely reachable automata [14, Problem 4] and another general one, in relation to avoiding words [12, Conjecture 15].

1.1 Contribution

We design a polynomial-time algorithm for the problem of deciding whether an automaton is completely reachable, thus solving the 7-year-old open question [6]. For this solution, we develop a complement-intersecting technique, which allows finding a short extending word for a given subset of states (i.e., that gives a larger preimage). We optimize the complexity of the algorithm to work in $\mathcal{O}(|\Sigma| \cdot n^3)$ time.

Based on the discovered properties, we prove that every non-empty subset of $k < n$ states in a completely reachable n-state automaton is reachable with a word of length smaller than $2n(n-k)$. This is a weaker version (by the factor of 2) of Don's conjecture stated for completely reachable automata [14, Problem 4].

It follows that a completely reachable n-state automaton has a reset word of length at most $2n^2 - n \ln n - 4n + 2$ (for $n \geq 3$). This generalizes and improves the previous bounds obtained with different techniques for known proper subclasses of completely reachable automata: automata with a full transition monoid (with the previous upper bound $2n^2 - 6n + 5$) [13], synchronizing automata with simple idempotents (with the previous upper bound $2n^2 - 4n + 2$) [19], and 1-contracting automata (only a special case was solved) [10].

2 Solving Complete Reachability in Polynomial Time

2.1 Reachability

A *complete deterministic finite semiautomaton* (called simply *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 the *transition function*, which is extended to a function $Q \times \Sigma^* \to Q$ in the usual way. Throughout the paper, by n we always denote the number of states in Q.

Given a subset $S \subseteq Q$, the *image* of S under the action of a word $w \in \Sigma^*$ is $\delta(S, w) = \{\delta(q, w) \mid q \in S\}$. The *preimage* of S under the action of w is $\delta^{-1}(S, w) = \{q \in Q \mid \delta(q, w) \in S\}$. For a state q, we also simplify $\delta^{-1}(q, w) = \delta^{-1}(\{q\}, w)$. Note that $q \in \delta(Q, w)$ if and only if $\delta^{-1}(q, w) \neq \emptyset$. For a subset $S \subseteq Q$, by $\overline{S}$ we denote its complement $Q \setminus S$.

For two subsets $S, T \subseteq Q$, if there exists a word $w \in \Sigma^*$ such that $\delta(T, w) = S$, then we say that S is *reachable from* T *with* the word w. Then we also say that T is a *w-predecessor* of S. It is simply a *predecessor* of S if it is a w-predecessor for some word w. One set can have many w-predecessors, but there is at most one maximal with respect to inclusion (and size).

▶ Remark 1. For $S \subseteq Q$ and $w \in \Sigma^*$, the preimage $\delta^{-1}(S, w)$ is a w-predecessor of S if and only if $\delta^{-1}(q, w) \neq \emptyset$ for every state $q \in S$. Equivalently, we have $\delta(\delta^{-1}(S, w)) = S$.

▶ Remark 2. For $S \subseteq Q$ and $w \in \Sigma^*$, if $\delta^{-1}(S, w)$ is a w-predecessor of S, then all w-predecessors of S are contained in it, thus $\delta^{-1}(S, w)$ is maximal in terms of inclusion and size. If $\delta^{-1}(S, w)$ is not a w-predecessor of S, then S does not have any w-predecessors.

A word w is called *extending* for a subset S, or w *extends* S, if $|\delta^{-1}(S, w)| > |S|$. It is called *properly extending* if additionally $\delta^{-1}(S, w)$ is a w-predecessor of S, i.e., $\delta^{-1}(q, w) \neq \emptyset$ for all $q \in S$.

A subset $S \subseteq Q$ is *reachable* in the automaton if S is reachable from Q with any word. An automaton is *completely reachable* if all non-empty subsets $S \subseteq Q$ are reachable. Equivalently, Q is a predecessor of all its non-empty subsets. The latter leads to an alternative characterization of completely reachable automata:

▶ Remark 3. An automaton (Q, Σ, δ) is completely reachable if and only if for every non-empty proper subset of Q, there is a properly extending word.

The decision problem COMPLETELY REACHABLE is the following: Given an automaton (Q, Σ, δ), is it completely reachable?

2.2 Witnesses

For an automaton (Q, Σ, δ), we consider unreachable sets that have the maximal size among all unreachable subsets of Q. They play the role of witnesses for the non-complete reachability of the automaton (or counterexamples for its complete reachability).

▶ **Definition 4** (Witness). *A non-empty set $S \subsetneq Q$ is a* witness *if it is unreachable and has the maximal size of all unreachable subsets of Q.*

Since Q is trivially reachable, the maximal size of unreachable subsets of Q is in $\{0, \dots, n-1\}$. It equals 0 (i.e., there is no witness) if and only if the automaton is completely reachable.

Although any non-empty unreachable set is evidence that the automaton is not completely reachable, it turns out that verifying if a set is a witness is computationally easier – later we show that we can verify the complete reachability and also find a witness if it exists in polynomial time. Verifying whether a given set is reachable (or unreachable) in general is PSPACE-complete [6] and it remains hard in many variations of the problem [3].

We start with a simpler solution in co-NP, where we just guess a candidate for a witness and verify it. A witness obviously cannot have a larger predecessor, as this predecessor or maybe some other larger set would be unreachable and so a witness instead. Still, a set can have exponentially many predecessors of the same size. The following observation allows us to infer the existence of a larger predecessor indirectly.

▶ **Lemma 5.** *Let $S, T \subseteq Q$ be distinct. If a word $u \in \Sigma^*$ is properly extending $S \cup T$, then u is also properly extending S or T.*

Proof. Let u be a properly extending word for $U = S \cup T \subsetneq Q$, thus there is a larger maximal u-predecessor U' of U, i.e., $\delta(U', u) = U$, $\delta^{-1}(U, u) = U'$, and $|U'| > |U|$. By Remark 1, we have $|\delta^{-1}(q, u)| \geq 1$ for all $q \in U$. As this holds for all the states of S and T, they also have their u-predecessors $\delta^{-1}(S, u)$ and $\delta^{-1}(T, u)$, respectively. Suppose that $|\delta^{-1}(S, u)| = |S|$ and $|\delta^{-1}(T, u)| = |T|$. Then $|\delta^{-1}(q, u)| = 1$ for all $q \in U$, which gives a contradiction with $|U'| > |U|$. ◀

▶ **Corollary 6.** *Let $S, T \subsetneq Q$ be two distinct witnesses. Then $S \cup T = Q$.*

Proof. Since $S \neq T$, the union $S \cup T$ is larger than the witness size $|S| = |T|$, thus it must be reachable. If $S \cup T \neq Q$, then $S \cup T$ has a larger predecessor. By Lemma 5, either S or T also has a larger predecessor, which contradicts that they both are witnesses. ◀

Now, we focus on detecting sets that are potential witnesses. We relax the required property for being a witness and introduce an auxiliary definition:

▶ **Definition 7** (Witness candidate). *A non-empty set $S \subsetneq Q$ is a* witness candidate *if it does not have a larger predecessor and all its predecessors (which are of the same size) have pairwise disjoint complements.*

▶ Remark 8. For two sets $T, T' \subseteq Q$, the condition of disjoint complements $\overline{T} \cap \overline{T'} = \emptyset$ is equivalent to $T \cup T' = Q$.

▶ **Lemma 9.** *A witness is a witness candidate.*

Proof. Let S be a witness and let T and T' be two distinct predecessors of S. As they are of the same size $|T| = |T'| = |S|$ and S is reachable from both T and T', they also must be witnesses. By Corollary 6, $T \cup T' = Q$. Therefore, S meets the definition of a witness candidate. ◀

Thus, every witness is a witness candidate and clearly, every witness candidate is unreachable (moreover, it is unreachable from every larger set). However, both converses do not necessarily hold.

▶ **Example 10.** The automaton from Figure 1 (left) is not completely reachable, and:
- All sets of size 5 are witnesses.
- The sets $Q \setminus \{q_i, q_{(i+1) \bmod 6}\}$ for $i \in \{0, \dots, 5\}$ are reachable.

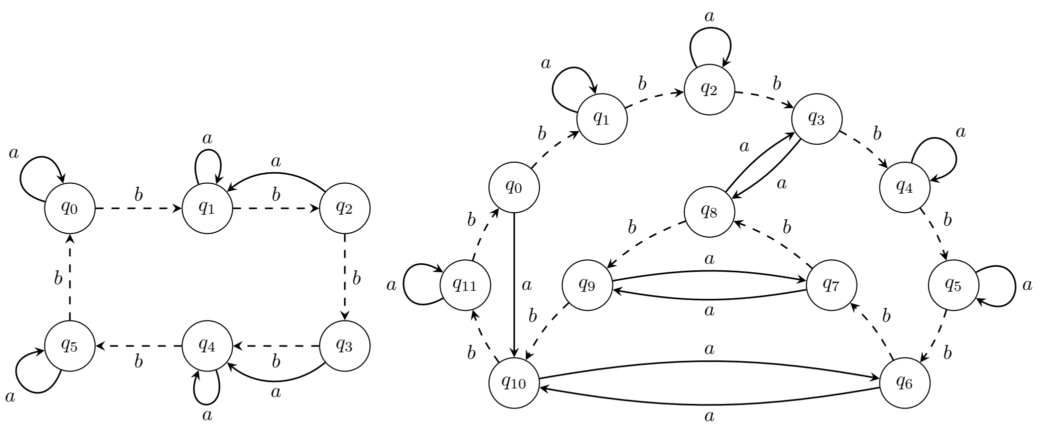

Figure 1 Left: an example of an automaton that is not completely reachable. **Right:** an example of a completely reachable automaton [8, Fig. 3]. (a's transitions are solid, b's transitions are dashed.)

- The sets $Q \setminus \{q_i, q_{(i+2) \bmod 6}\}$ for $i \in \{0, \dots, 5\}$ are unreachable (only the action of b yields a predecessor). They do not admit a properly extending word but they are not witness candidates, since their complements are not disjoint.
- The set $\{q_0, q_2, q_4\}$ and its complement $\{q_1, q_3, q_5\}$ are witness candidates but not witnesses, as they have size 3.
- All non-empty proper subsets of Q are extensible, e.g., $\delta^{-1}(q_1, (ab)^3, a) = Q$, but not necessarily properly extensible.

The following remark will be useful for efficient checking if a set is a witness candidate:

▶ Remark 11. A witness candidate S has at most $\lfloor n/(n - |S|) \rfloor$ predecessors.

Proof. The complements of the predecessors of S are pairwise disjoint, so each state is contained in at most one complement of size $n - |S|$. ◀

2.3 An Algorithm in co-NP

We build a polynomial procedure that checks whether a given set S is a witness candidate. It is shown in Algorithm 1. Starting from S, we process all its predecessors in a breath-first search manner; for this, a FIFO queue *Process* is used. A next set T is taken in line 6. Then, we verify whether $\overline{T} \cap \overline{T'} \neq \emptyset$, for some previously processed set T'. For this, we maintain *Absent* array, which for every state q indicates whether q occurred in the complement of some previously processed set, and if so, this set is stored as *Absent*$[q]$. We additionally use this array to check whether the same set T has been processed previously (line 9); if so, then it is ignored. Otherwise, S is not a witness candidate as the complements of its two predecessors have a non-empty intersection. We update this array in lines 12–13. Finally, in lines 14–19, we add one-letter predecessors of T to the queue. If one of the predecessors is larger than T, then this immediately implies that S is not a witness candidate. Since predecessors are never smaller, this means that all processed sets that are put into the queue are of the same size $|S|$. When all predecessors of S have been considered and neither of the two conditions occurred, the function reports that S is a witness candidate.

The function is a base for our next algorithms. Hence, even though we do not need to optimize it here, in the next lemma, we describe the technical details for achieving optimal time complexity. In particular, an important optimization that lowers the time complexity

Algorithm 1 Verifying whether a given subset is a witness candidate from Definition 7.

```
Input: An $n$-state automaton $\mathscr{A} = (Q, \Sigma, \delta)$ and a non-empty $S \subsetneq Q$.
Output: true if $S$ is a witness candidate; false otherwise.
 1: function IsWitnessCandidate($\mathscr{A}, S$)
 2:     $Process \leftarrow$ EmptyFifoQueue()     ▷ It contains predecessors of $S$ to be processed
 3:     $Process$.Push($S$)
 4:     $Absent \leftarrow$ Array indexed by $q \in Q$ initialized with none
 5:     while not $Process$.Empty() do
 6:         $T \leftarrow Process$.Pop()
 7:         if $Absent[q] \neq$ none for some $q \in \overline{T}$ then      ▷ Then $T \cup T' \neq Q$
 8:             $T' \leftarrow Absent[q]$
 9:             if $T = T'$ then
10:                 continue                              ▷ $T$ has been processed previously
11:             return false      ▷ $T$ and $T'$ are predecessors with non-disjoint complements
12:         for all $q \in \overline{T}$ do
13:             $Absent[q] \leftarrow T$
14:         for all $a \in \Sigma$ do
15:             if $T$ has $a$-predecessor then
16:                 $T' = \delta^{-1}(T, a)$
17:                 if $|T'| > |S|$ then
18:                     return false
19:                 $Process$.Push($T'$)
20:     return true                                   ▷ All predecessors of $S$ were checked
```

is storing each processed set T as a list of states in its complement. Then we can perform all operations on a set T in $\mathcal{O}(n - |S|)$ time, which compensates the number of iterations, which is at most $\mathcal{O}(n/(n - |S|))$.

▶ **Lemma 12.** *Function* IsWitnessCandidate *from Algorithm 1 is correct and can be implemented to work in $\mathcal{O}(|\Sigma| \cdot n)$ time.*

Proof. *Correctness:* The function can end in line 11, 18, or 20. The first case means that S has two distinct predecessors, T and T' such that $\overline{T} \cap \overline{T'} \neq \emptyset$, which implies that S is not a witness candidate. The second case (line 18) means that we have found a larger predecessor of S, thus S is not a witness candidate. The last possibility (line 20), where the function ends with a positive answer, occurs when there are no more predecessors of S to consider (*Process* becomes empty), so all predecessors of S have been checked and the two previous cases have not occurred. Thus, S satisfies the conditions from Definition 7.
Running time: We separately consider the time complexity of two types of iterations of the main loop: *full* and *extra* iterations. An extra iteration is where the case from line 7 holds; then the iteration ends either in line 10 or 11. Otherwise, we count it as a full iteration.

In each iteration, T has size $|S|$, because a predecessor cannot be smaller than the set and we check in line 18 if it is larger. The number of distinct processed predecessors (including S itself) of the same size $|S| < n$ with pairwise disjoint complements is at most $\lfloor n/(n - |S|) \rfloor$. This bounds the number of full iterations. In extra iterations, the same sets can be repeated. As these sets are added in full iterations, and one full iteration adds at most $|\Sigma|$ sets to the queue, the number of extra iterations is at most $|\Sigma| \cdot (\lfloor n/(n - |S|) \rfloor)$.

If we store T (and all sets in *Process*) as a list of states in its complement, then full one iteration takes $\mathcal{O}(|\Sigma| \cdot (n-|S|))$ time: Lines 6–11 trivially take $\mathcal{O}(n-|S|)$ time. Updating *Absent* in lines 12–13 also takes $\mathcal{O}(n-|S|)$ time, if we store a pointer/reference to T (line 14 in $\mathcal{O}(1)$) instead of copying. Computing one-letter predecessors in lines 15–19 can be performed in $\mathcal{O}(|\Sigma| \cdot (n-|S|))$ time as follows. At the beginning of the function, we additionally do some preprocessing of the automaton. For each $a \in \Sigma$ and $q \in Q$, we compute the list of states $\delta^{-1}(q,a)$; note that for the same a, these lists are always disjoint. For each $a \in \Sigma$, we also count the states $q \in Q$ such that $|\delta^{-1}(q,a)| = 0$; let z_a be their number. Then in line 15, we check if $\delta^{-1}(T,a)$ is a-predecessor by counting the states $p \in \overline{T}$ such that $|\delta^{-1}(p,a)| = 0$. The set $\delta^{-1}(T,a)$ is a-predecessor if and only if the number of such states p equals z_a, because if it is smaller, then some state $q \in T$ has an empty preimage under the action of a. Next, we compute $\delta^{-1}(\overline{T},a)$ by joining the preprocessed lists for a for each $q \in \overline{T}$, and $T' = Q \setminus \delta^{-1}(\overline{T},a)$ is also stored in the form of a list of states in the complement. Since a predecessor is never smaller than the set, we have $|T'| \geq |T| = |S|$, thus the length of this list is at most $|S|$, so we do this computation in $\mathcal{O}(n-|S|)$ time.

Also, the running time of an extra iteration (lines 6 up to 11) is easily bounded by $\mathcal{O}(n-|S|)$ time.

Summarizing, full iterations take $\mathcal{O}(n/(n-|S|) \cdot |\Sigma| \cdot (n-|S|))$ time and extra iterations take $\mathcal{O}(|\Sigma| \cdot n/(n-|S|) \cdot (n-|S|))$ time, which gives the same upper bound on the total. The aforementioned preprocessing can be done in $\mathcal{O}(|\Sigma| \cdot n)$ time as well. ◀

▶ Remark 13. The asymptotic running time of IsWitnessCandidate in terms of $|\Sigma|$ and n is the best possible because $|\Sigma| \cdot n$ is the number of automaton's transitions that we have to read in the worst case.

▶ **Theorem 14.** *Problem Completely Reachable can be solved in co-NP.*

Proof. To certify that a given automaton $\mathscr{A}$ is not completely reachable, we can guess a witness candidate $S \subsetneq Q$ and call IsWitnessCandidate($\mathscr{A}, S$) to verify it. If the automaton is not completely reachable, then there exists some witness, which is a witness candidate. Otherwise, there are no unreachable non-empty sets, thus no witness candidates. ◀

2.4 A Polynomial-Time Algorithm

The overall idea to make the algorithm work in deterministic polynomial time is as follows. We replace guessing a witness with a constructive procedure. We extend the function from Algorithm 1 so that instead of a Boolean answer, it finds a properly extending word for S. This works under a certain assumption that S is not a witness candidate and there are no witness candidates of size larger than $|S|$. When S is a witness candidate, the function returns **none**.

Then, we use this function to hunt for a witness. We iteratively reduce each set of size $n-1$ to smaller sets S in the way that if some witness is a subset of the initial set, then this witness is also a subset of set S. As we process the sets from the largest size, we keep the assumption that there are no witness candidates larger than the currently processed set. Hence, the first found witness candidate will be a witness.

2.4.1 Finding Properly Extending Words

The function for finding properly extending words is shown in Algorithm 2. Here, together with predecessor sets T of S, we also keep track of the words w such that $\delta(T,w) = S$. The main difference with IsWitnessCandidate is the case in line 7. For the union $T \cup T'$,

Algorithm 2 An algorithm finding a properly extending word (a larger predecessor) for a given set (recursive version).

```
Input: An $n$-state automaton $\mathscr{A} = (Q, \Sigma, \delta)$ and a non-empty $S \subsetneq Q$.
Output: A properly extending word for $S$ or none . If $S$ is a witness candidate, then always
    returns none . If $S$ is not a witness candidate and there are no witness candidates
    of size $> |S|$, then always returns a word.
1:  function FindProperlyExtendingWord($\mathscr{A}$, $S$)
2:      $Process \leftarrow$ EmptyFifoQueue()    ▷ It contains pairs $(T, w)$ such that $\delta(T, w) = S$
3:      $Process$.Push($(S, \varepsilon)$)
4:      $Absent \leftarrow$ Array indexed by $q \in Q$ initialized with none
5:      while not $Process$.Empty() do
6:          $(T, w) \leftarrow Process$.Pop()
7:          if $Absent[q] \neq$ none for some $q \in \overline{T}$ then    ▷ Then $T \cup T' \neq Q$
8:              $(T', w') \leftarrow Absent[q]$
9:              if $T' = T$ then
10:                 continue    ▷ $T'$ has been processed previously
11:             $u \leftarrow$ FindProperlyExtendingWord($\mathscr{A}$, $T \cup T'$)
12:             if $u =$ none then
13:                 return none
14:             else    ▷ $u$ is a properly extending word for $T \cup T'$
15:                 if $|\delta^{-1}(T, u)| > |T|$ then    ▷ $u$ properly extends $T$
16:                     return $uw$
17:                 else    ▷ $u$ properly extends $T'$
18:                     return $uw'$
19:         for all $q \in \overline{T}$ do
20:             $Absent[q] \leftarrow (T, w)$
21:         for all $a \in \Sigma$ do
22:             if $T$ has $a$-predecessor then
23:                 $T' = \delta^{-1}(T, a)$
24:                 if $|T'| > |S|$ then
25:                     return $aw$
26:                 $Process$.Push($(T', aw)$)
27:     return none    ▷ $S$ is a witness candidate
```

we aim at finding a properly extending word for it, which then turns out to be properly extending either for T or T' by Lemma 5; this is done by a recursive call in line 11. This call can return **none** instead, which means that a witness candidate larger than S was found.

▶ **Example 15.** For the automaton from Figure 1 (left) and $S = \{q_0, q_2, q_4, q_5\}$ (this is an unreachable and not properly extensible set, but not a witness candidate), FindProperlyExtendingWord returns **none** using one recursive call.

Proof. Since letter a cannot be applied, the second iteration is with $T = \delta^{-1}(S, b) = \{q_5, q_1, q_3, q_4\}$, and similarly, the third one is with $T = \delta^{-1}(S, b^2) = \{q_4, q_0, q_2, q_3\}$. Now, since the complements of $\{q_4, q_0, q_2, q_3\}$ and S contain the common state q_1, we recursively call the function for the union $Q \setminus \{q_1\}$. This union set is a witness and has 6 predecessors (including itself). After processing all these predecessors, the function returns **none**. ◀

▶ **Example 16.** For the automaton from Figure 1 (right) and let $S = \{q_0, q_{10}\}$, FindProperlyExtendingWord returns the word ab^2 using 8 recursive calls.

▶ **Lemma 17.** *Function FindProperlyExtendingWord is correct and works in at most $\mathcal{O}(|\Sigma| \cdot n^2 \log n)$ time. Moreover, if a word is returned, then it has length at most $\mathcal{O}(n \log n)$.*

Proof sketch. The correctness can be proved by descending induction on the set size, using similar arguments as for Lemma 12 and Lemma 5

Without counting the recursive call, the running time is $\mathcal{O}(|\Sigma| \cdot n^2/(n - |S|)$; for this, to avoid higher complexity by copying potentially long words, we need to concatenate words by storing pointers to both parts and maintain the induced transformations $Q \to Q$ along them. If we consider the recursive calls, in the worst case $n - |S|$ we call the function on sets of sizes $n - |S|, n - |S| + 1, \dots, n - 1$; in the calculation we get the harmonic series what is bounded by the logarithm, giving the final running time and word length bounds. ◀

2.4.2 Set Reduction for Witness Containment

Suppose that given a subset $S \subsetneq Q$, we search for a witness that is contained within it. Having a properly extending word w for S, we can reduce S to its proper subset by excluding some states which surely cannot be in any witness contained in S. The next lemma shows the criterion.

▶ **Lemma 18.** *Let $S \subseteq Q$ and $\delta^{-1}(S, w)$ be a w-predecessor of S for some w. Then for every witness candidate $S' \subseteq S$, every state $q \in S'$ is such that $|\delta^{-1}(q, w)| = 1$.*

Proof. Since $\delta^{-1}(S, w)$ is a w-predecessor of S, by Remark 1, we have $\delta^{-1}(q, w) \neq \emptyset$ for every state $q \in S$. Suppose for a contradiction that there exists $S' \subseteq S$ that is a witness candidate and contains a state $p \in S'$ such that $|\delta^{-1}(p, w)| > 1$. Note that the set $\delta^{-1}(S', w)$ is a w-predecessor of S', since its superset S has a w-predecessor, and:

$$|\delta^{-1}(S', w)| = |\delta^{-1}(p, w)| + \sum_{s \in S \setminus \{p\}} |\delta^{-1}(s, w)| > 1 + (|S| - 1) = |S|.$$

Thus, S' cannot be a witness candidate, since it has a larger predecessor. ◀

As a witness is also a witness candidate, the lemma also applies to witnesses. From the lemma, we know that by having a properly extending word for S, we can remove at least one state from S and all the witnesses will still be contained in the resulting set. Function Reduce$(\mathscr{A}, S, w)$ in Algorithm 3 realizes this reduction and returns a set of states that can be removed. If w is given as a transformation, the function trivially works in $\mathcal{O}(n)$ time.

Algorithm 3 Reducing a set for possible witness containment.

```
Input: An $n$-state automaton $\mathscr{A} = (Q, \Sigma, \delta)$, a non-empty $S \subsetneq Q$, and a properly extending
    word $w$ for $S$.
Output: A non-empty subset $R \subseteq S$ such that all the witnesses contained in $S$ are also
    contained in $S \setminus R$.
1: function Reduce($\mathscr{A}$, $S$, $w$)
2:     return $\{p \in S : |\delta^{-1}(p, w)| > 1\}$
```

2.4.3 Finding a Witness

We have all ingredients to build a polynomial algorithm that solves the decision problem Completely Reachable. It is shown in Algorithm 4. If the automaton is not completely reachable, the algorithm also finds a witness. Starting from all sets of size $n-1$, we process sets in descending order by size. Processing a set consists of finding a properly extending word for it and reducing the set for a witness containment by this word.

Algorithm 4 A polynomial-time algorithm verifying the complete reachability of an automaton or finding a witness.

```
Input: An $n$-state automaton $\mathscr{A} = (Q, \Sigma, \delta)$.
Output: none if $\mathscr{A}$ is completely reachable; a witness otherwise.
 1: function FindWitness($\mathscr{A}$)
 2:     $Queue \leftarrow$ EmptyPriorityQueue()   ▷ Ordered by set size; the largest sets go first
 3:     for all $q \in Q$ do
 4:         $Queue$.Push($Q \setminus \{q\}$)   ▷ Initialize with sets of size $n-1$
 5:     while not $Queue$.Empty() do
 6:         $S \leftarrow Queue$.Pop()   ▷ Get a set of the largest size
 7:         $w \leftarrow$ FindProperlyExtendingWord($\mathscr{A}, S$)
 8:         if $w =$ none then
 9:             return $S$   ▷ Found witness
10:         $R \leftarrow$ Reduce($\mathscr{A}, S, w$)   ▷ $R$ is non-empty
11:         $S' \leftarrow S \setminus R$   ▷ $S' \subsetneq S$
12:         if $S' \neq \emptyset$ then
13:             $Queue$.Push($S'$)
14:     return none   ▷ No witnesses
```

▶ **Lemma 19.** *Function FindWitness is correct and works in $\mathcal{O}(|\Sigma| \cdot n^4 \log n)$ time.*

▶ Remark 20. The number of witnesses of size k is at most $\lfloor n/k \rfloor$, as their complements must be pairwise disjoint – otherwise, we could properly extend at least one of them by Lemma 5. Function FindWitness can be easily modified to find all the witnesses: after finding the first witness of size k, we can continue the main loop until all sets of size k are processed.

2.5 Improving Running Time

The main idea for the improvement is to use already computed reductions instead of finding a properly extending word recursively (Algorithm 2, line 13), which is required when we encounter the case of non-disjoint complements. For lowering the time complexity by this optimization, using adequate set representations is also crucial.

2.5.1 Reduction History

A *reduction history RED* is an array of size n of lists of states. For a state $q \in Q$, $RED[q]$ denotes the list of states assigned to q. Let $|RED[q]|$ denote the length of this list, and let $RED[q][i]$ denote the i-th state for $1 \leq i \leq |RED[q]|$. The states in the list must be pairwise different and distinct from q.

A reduction history represents our current knowledge about possible witness containment and will be progressively filled out in the algorithm, starting from the empty lists. For each $q \in Q$, the reduction history defines a *reduction chain* that is a sequence of reduced sets

$R^q_0, R^q_1, \dots, R^q_{|RED[q]|}$. The first set $R^q_0 = Q \setminus \{q\}$, and each next set is obtained from the previous set by removing the corresponding state in the list: for $1 \le i \le |RED[q]|$, we define $R^q_i = R^q_{i-1} \setminus \{RED[q][i]\}$ (equivalently, $R^q_i = Q \setminus (\{q\} \cup \bigcup_{i \in \{1,\dots,|RED[q]|\}} \{RED[q][i]\})$).

A reduction history RED is *valid* (for an automaton) if for each $q \in Q$ and each $1 \le i \le |RED[q]|$, there exists a properly extending word w for R^q_{i-1} such that $|\delta^{-1}(RED[q][i], w)| > 1$. This means that the reduction for witness containment of R^q_{i-1} to R^q_i by removing the state $RED[q][i]$ is justified by Lemma 18, i.e., all witnesses contained in R^q_{i-1} are also contained in R^q_i. However, note that the reduction can be not exhaustive with respect to w, i.e., there may exist other states $p \in R^q_{i-1} \setminus \{RED[q][i]\}$ such that $|\delta^{-1}(p, w)| > 1$, which also could be removed by Lemma 18. In our algorithm, this situation will be possible because we do not always compute properly extending words directly but infer their existence based on the reductions computed for other sets, tracing only one state to remove.

Besides being valid, we need that our reduction history is sufficiently filled out. For $q \in Q$, the *deficiency* of $RED[q]$ is the length $|RED[q]|$ thus equals the number of states to be removed from $Q \setminus \{q\}$. The *deficiency* of the whole reduction history RED is its minimum deficiency over $q \in Q$. Hence, a valid history reduction of deficiency d stores information for reducing every set of size $n - 1$ to a set of size at most $n - 1 - d$.

Algorithm 5 A fast reduction retrieval for witness containment for a given set from the past stored reductions.

Input: An n-state automaton $\mathscr{A} = (Q, \Sigma, \delta)$, a non-empty set $S \subsetneq Q$, and a reduction history RED.
Require: RED is a valid reduction history of deficiency at least $n - |S|$.
Output: A state $p \in S$ such that there exists a properly extending word w for S such that $|\delta^{-1}(p, w)| > 1$.

```
1: function GetStoredReduction($\mathscr{A}$, $S$, $RED$)
2:     $q \leftarrow$ any state from $\overline{S}$
3:     for $i \leftarrow 1, 2, \dots$ do
4:         assert($i \le |RED[q]|$)
5:         $p \leftarrow RED[q][i]$
6:         if $p \in S$ then
7:             return $p$
```

Function GetStoredReduction from Algorithm 5 quickly finds a reduction of a given set S, provided that a valid reduction history with a large enough deficiency is available. The function starts from picking any q outside of S, and it repeats the reduction chain starting from $Q \setminus \{q\} \supseteq S$. It seeks the first removed state that belongs to S. In this way, since the same reduction was applied previously to a superset of S, it can be correctly applied to S as well, i.e., the same word is properly extending word for both the superset and S. We note that this may not hold for the states removed later, since then a properly extending word inferred from the reduction history may not be properly extending for S (may not give a predecessor of S).

Since the found state $p \in S$ has the property that there exists a properly extending word w for S such that $|\delta^{-1}(p, w)| > 1$, we can apply Lemma 18 and get a smaller $S' = S \setminus \{p\} \subsetneq S$ such that every witness in S is also contained in S'.

▶ **Lemma 21.** *Function* GetStoredReduction *is correct and can be implemented to work in $\mathcal{O}(n)$ time.*

2.5.2 Finding a Reduction

We redesign earlier Algorithm 2 for finding a properly extending word so that we return a reduction (here, one state to remove) directly for the given S. GetStoredReduction can be used for computing the reduction fast, but only if our reduction history has a large enough deficiency, i.e., in each of the reduction chains, $Q \setminus \{q\}$ is reduced to a set smaller than the set that we want to reduce. We cannot ensure this for S, but if in the main algorithm, we reduce the sets in descending order by their size, then we can fulfil the weaker requirement that the reduction chains end with sets of size at most $|S|$. Then, to reduce S, we perform as the previous algorithm until encountering the case of a non-empty complements intersection. Then we can use GetStoredReduction for the obtained set of size at least $|S|+1$.

Function FindReduction from Algorithm 6 for a given S finds a state to remove or **none** if S is a witness candidate. The most important difference with previous FindProperlyExtendingWord is the use of GetStoredReduction in line 11 and processing its result p.

To keep the time complexity low, as before, we need to store the sets T in the form of a list of states in the complement. This time, we do not maintain the induced transformations for words (they are too costly here). Instead, we compute $\delta(p, w)$ applying the letters one by one, but only for this state, which is doable in $\mathcal{O}(n/(n-|S|))$ time, avoiding quadratic time complexity.

▶ **Lemma 22.** *Function FindReduction is correct and can be implemented to work in $\mathcal{O}(|\Sigma| \cdot n)$ time.*

2.5.3 The Optimized Algorithm

The optimized algorithm FindWitnessFaster is shown in Algorithm 7. For reducing sets, it relies on FindReduction, which returns a state p such that there exists a properly extending word w for S and $|\delta^{-1}(p, w)| > 1$. We can remove p from S by Lemma 18, i.e., every witness contained in S must be also contained in $S' = S \setminus \{p\}$.

Additionally, the sets in *Queue* are stored together with their initially removed state q, to know where to store their reductions in the reduction history. Note that in an iteration, the set S taken from the queue is the currently last set $R^q_{|RED[q]|}$ from the reduction chain for q, and S' will be a next set in this chain. Updating the reduction history with state p keeps it valid, which also follows from the guaranteed existence of a properly extending word by FindReduction. As we process the sets in descending order by size, our reduction history always has the required deficiency when used in line 8.

▶ **Lemma 23.** *Function FindWitnessFaster is correct and works in $\mathcal{O}(|\Sigma| \cdot n^3)$ time.*

3 An Upper Bound on Reset Threshold

3.1 Synchronization

A *reset word* is a word w such that $|\delta(Q, w)| = 1$. Equivalently, we have $\delta^{-1}(q, w) = Q$ for some $q \in Q$. If an automaton admits a reset word, then it is called *synchronizing* and its *reset threshold* is the length of the shortest reset words.

The central problem in the theory of synchronizing automata is the famous Černý conjecture, which states that every synchronizing n-state automaton has its reset threshold at most $(n-1)^2$. For the subclass of completely reachable automata, the previously known upper bound on the reset threshold was $7/48n^3 + \mathcal{O}(n^2)$ [5], which has been obtained through the technique of avoiding [22], that is, it follows in particular from the fact that every set of size $n-1$ is reachable with a word of length at most n.

Algorithm 6 An algorithm finding a reduction for witness containment using a reduction history.

```
Input: An $n$-state automaton $\mathscr{A} = (Q, \Sigma, \delta)$, a non-empty $S \subsetneq Q$, and a reduction history
    $RED$.
Require: $RED$ is a valid reduction history of deficiency at least $n - |S| - 1$.
Output: If $S$ is not a witness candidate: a state $p \in S$ such that there exists a properly
    extending word $w$ for $S$ such that $|\delta^{-1}(p, w)| > 1$. Otherwise: none.
 1: function FindReduction($\mathscr{A}, S, RED$)
 2:     $Process \leftarrow$ EmptyFifoQueue()
 3:     $Process$.Push(($S, \varepsilon$))
 4:     $Absent \leftarrow$ Array indexed by $q \in Q$ initialized with none
 5:     while not $Process$.Empty() do
 6:         $(T, w) \leftarrow Process$.Pop()
 7:         if $Absent[q] \neq$ none for some $q \in \overline{T}$ then        ▷ Then $T \cup T' \neq Q$
 8:             $(T', w') \leftarrow Absent[q]$
 9:             if $T' = T$ then
10:                 continue                    ▷ $T'$ has been processed previously
11:             $p \leftarrow$ GetStoredReduction($\mathscr{A}, T \cup T', RED$)
12:             if $p \in T$ then
13:                 return $\delta(p, w)$
14:             else                                                   ▷ Then $p \in T'$
15:                 return $\delta(p, w')$
16:         for all $q \in \overline{T}$ do
17:             $Absent[q] \leftarrow (T, w)$
18:         for all $a \in \Sigma$ do
19:             if $T$ has $a$-predecessor then
20:                 $T' = \delta^{-1}(T, a)$
21:                 if $|T'| > |S|$ then
22:                     $p \leftarrow$ any state such that $|\delta^{-1}(T, a)| > 1$
23:                     return $\delta(p, w)$
24:                 $Process$.Push(($T', aw$))
25:     return none                                    ▷ $S$ is a witness candidate
```

3.2 Finding Short Properly Extending Words

For a completely reachable automaton, function FindProperlyExtendingWord from Algorithm 8 always finds a properly extending word. Therefore, using the well-known *extension* method (e.g., [24]), we can construct a synchronizing word starting from some singleton $\{q\}$ and iteratively increasing the set by at least one in at most $n - 1$ iterations, finally obtaining Q. This is an easy way to get the upper bound of order $\mathcal{O}(n^2 \log n)$ by Lemma 17. However, it is not enough to prove a quadratic upper bound, so we are going to further adapt the algorithm for that.

The idea is to keep track of all subsets for an intersection of complements, instead of starting an independent search for a properly extending word recursively. This modification is shown in Algorithm 8.

The function keeps *Trace* map, which stores for a given predecessor set, by the application of what letter it has been obtained or, in the second case, of what two sets it is a union. It also stores S' as the current origin set, which is the set for which we are going to find a larger predecessor currently.

Algorithm 7 A faster algorithm for finding a witness.

```
Input: An $n$-state automaton $\mathscr{A} = (Q, \Sigma, \delta)$.
Output: none if $\mathscr{A}$ is completely reachable; a witness otherwise.
 1: function FindWitnessFaster($\mathscr{A}$)
 2:     $RED[q] \leftarrow$ EmptyList  for all $q \in Q$          ▷ Empty reduction history
 3:     $Queue \leftarrow$ EmptyPriorityQueue()     ▷ Contains pairs $(S, q)$; ordered by $|S|$
 4:     for all $q \in Q$ do
 5:         $Queue$.Push($(Q \setminus \{q\}, q)$)
 6:     while not $Queue$.Empty() do
 7:         $(S, q) \leftarrow Queue$.Pop()               ▷ Get a set of the largest size
 8:         $p \leftarrow$ FindReduction($\mathscr{A}, S, RED$)
 9:         if $p =$ none then
10:             return $S$                                       ▷ Found witness
11:         $RED[q]$.append($p$)                          ▷ Store the removed state
12:         $S' \leftarrow S \setminus \{p\}$
13:         if $S' \neq \emptyset$ then
14:             $Queue$.Push($(S', q)$)
15:     return none                                               ▷ No witnesses
```

The function has two phases. First, it searches for a larger predecessor or a non-empty complement intersection as the previous variants. In the second case, it only changes the origin set S' (line 15), notes this in the map *Trace* and initiates a fresh search for S'. In the second phase (from line 25), a properly extending word is reconstructed (like in FindProperlyExtendingWord after a recursive call) until we reach our original S.

▶ **Lemma 24.** *Function FindShortProperlyExtendingWord is correct.*

The remaining effort is to prove an upper bound on the length of the returned word.

3.2.1 Nested Boxes

To bound the length of the found word, we consider an auxiliary combinatorial problem. Consider an n-element universe Q. Two subsets $S, T \subseteq Q$ are *colliding* if $S \cap T \notin \{\emptyset, S, T\}$. Thus, colliding sets have a non-trivial intersection. A family of non-empty subsets of Q is called *non-colliding* if all the subsets are pairwise non-colliding.

▶ **Definition 25.** *For an $n \geq 1$, the number $MaxNestedBoxes(n)$ is the maximum size of a non-colliding family for an n-element universe.*

The problem is equivalent to, e.g., the maximum number of boxes for n items, such that a box must contain either an item or at least two boxes.

▶ **Lemma 26.** $MaxNestedBoxes(n) = 2n - 1$.

The generalized version of the problem limits the maximum size of the subsets:

▶ **Definition 27.** *For an $n \geq 1$, the number $MaxNestedBoxes(n, k)$ is the maximum size of a non-colliding family for an n-element universe where each subset from the family has size at most k.*

▶ **Lemma 28.** $MaxNestedBoxes(n, k) = 2n - \lceil n/k \rceil$.

Algorithm 8 An algorithm finding a short properly extending word.

```
Input: An $n$-state automaton $\mathscr{A} = (Q, \Sigma, \delta)$ and a non-empty $S \subsetneq Q$.
Require: $\mathscr{A}$ is completely reachable.
Output: A properly extending word $w$ of $S$.
1:  function FindShortProperlyExtendingWord($\mathscr{A}, S$)
2:      $Trace \leftarrow$ EmptyMap()  ▷ For a processed set, it stores how this set was obtained; for not yet processed sets, it gives none
3:      $Process \leftarrow$ EmptyFifoQueue()
4:      $S' \leftarrow S$                                  ▷ Current origin set
5:      $Trace[S'] \leftarrow \varepsilon$
6:      $Process$.Push($S'$)
7:      while true do
8:          assert(not $Process$.IsEmpty())             ▷ Otherwise $S'$ is a witness candidate
9:          $T \leftarrow Process$.Pop()
10:         if $|T| > |S'|$ then                        ▷ Found a properly extending word for $S'$
11:             $S' \leftarrow T$
12:             break
13:         if there is $T'$ such that $Trace[T'] \neq$ none and $T \subsetneq T \cup T' \subsetneq Q$ then
14:             $S' \leftarrow T \cup T'$                   ▷ New origin set; $|S'| > |S|$
15:             $Trace[S'] \leftarrow (T, T')$
16:             $Process$.Clear()                       ▷ Continue only for the new origin
17:             $Process$.Push($S'$)
18:         else
19:             for all $a \in \Sigma$ do
20:                 if $a$ is properly extending $T$ then
21:                     $T' \leftarrow \delta^{-1}(T, a)$
22:                     if $Trace[T'] =$ none then       ▷ A not yet processed set
23:                         $Trace[T'] \leftarrow a$          ▷ $\delta(T', a) = T$
24:                         $Process$.Push($T'$)
25:     $w \leftarrow \varepsilon$                  ▷ Word reconstruction starts with the empty word
26:     while $S' \neq S$ do
27:         assert($Trace[S'] \neq$ none)
28:         if $Trace[S']$ is a letter then
29:             $a \leftarrow Trace[S']$
30:             $S' \leftarrow \delta(S', a)$
31:             $w \leftarrow wa$
32:         else
33:             $(T, T') \leftarrow Trace[S']$               ▷ $w$ properly extends $T \cup T'$
34:             if $|\delta^{-1}(T, w)| > |T|$ then          ▷ $w$ properly extends $T$
35:                 $S' \leftarrow T$
36:             else                                  ▷ $w$ properly extends $T'$
37:                 $S' \leftarrow T'$
38:     return $w$
```

3.2.2 Final Bounding

We apply the above combinatorial problem to derive an upper bound on the length of a word found by FindShortProperlyExtendingWord. The crucial property is that the family of the complements of all subsets T that are processed in the block of lines 19–24 is non-colliding, since for these subsets T, the condition in line 13 does not hold. The upper bound follows since all the letters in the final word are added in line 31, where S' is always the complement of one of the sets from the family.

▶ **Lemma 29.** *For a completely reachable automaton and a non-empty proper subset $S \subsetneq Q$, the word returned by* FindShortProperlyExtendingWord *from Algorithm 8 has length at most MaxNestedBoxes*$(n, n - |S|) = 2n - \lceil n/(n - |S|) \rceil$.

Finally, using the standard extension method (starting from a subset S and iteratively extending it to Q) and some calculations, we obtain a weaker Don's conjecture (cf. [14, Problem 4]):

▶ **Theorem 30.** *For a completely reachable n-state automaton (Q, Σ, δ), every non-empty proper subset $S \subseteq Q$ is reachable with a word of length at most*

$$(n - |S|)2n - n\ln(n - |S|) - n/(n - |S|) < 2n(n - |S|).$$

▶ **Corollary 31.** *The reset threshold of a completely reachable automaton with $n \geq 3$ states is at most*

$$(n-2)2n - n\ln(n-2) - n/(n-2) < 2n^2 - n\ln n - 4n + 2.$$

References

1 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.

2 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.

3 M. V. Berlinkov, R. Ferens, and M. Szykuła. Preimage problems for deterministic finite automata. *Journal of Computer and System Sciences*, 115:214–234, 2021.

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

5 E. A. Bondar, D. Casas, and M. V. Volkov. Completely reachable automata: an interplay between automata, graphs, and trees, 2022. `arXiv:2201.05075`.

6 E. A. Bondar and M. V. Volkov. Completely Reachable Automata. In Cezar Câmpeanu, Florin Manea, and Jeffrey Shallit, editors, *DCFS*, pages 1–17. Springer, 2016.

7 E. A. Bondar and M. V. Volkov. A Characterization of Completely Reachable Automata. In Mizuho Hoshi and Shinnosuke Seki, editors, *DLT*, pages 145–155. Springer, 2018.

8 D. Casas and M. V. Volkov. Binary completely reachable automata. In Armando Castañeda and Francisco Rodríguez-Henríquez, editors, *LATIN 2022: Theoretical Informatics*, pages 345–358. Springer, 2022. Full version at `arXiv:2205.09404`.

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

10 H. Don. The Černý Conjecture and 1-Contracting Automata. *Electronic Journal of Combinatorics*, 23(3):P3.12, 2016.

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

12 R. Ferens, M. Szykuła, and V. Vorel. Lower Bounds on Avoiding Thresholds. In *MFCS*, volume 202 of *LIPIcs*, pages 46:1–46:14. Schloss Dagstuhl, 2021.

13 F. Gonze, V. V. Gusev, B. Gerencser, R. M. Jungers, and M. V. Volkov. On the interplay between Babai and Černý's conjectures. In *DLT*, volume 10396 of *LNCS*, pages 185–197. Springer, 2017.

14 F. Gonze and R. M. Jungers. Hardly reachable subsets and completely reachable automata with 1-deficient words. *Journal of Automata, Languages and Combinatorics*, 24(2–4):321–342, 2019.

15 S. Hoffmann. Completely Reachable Automata, Primitive Groups and the State Complexity of the Set of Synchronizing Words. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *LATA*, LNCS, pages 305–317. Springer, 2021.

16 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.

17 M. Maslennikova. Reset complexity of ideal languages over a binary alphabet. *International Journal of Foundations of Computer Science*, 30(06n07):1177–1196, 2019.

18 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.

19 I.K. Rystsov. Estimation of the length of reset words for automata with simple idempotents. *Cybern. Syst. Anal. 36*, pages 339–344, 2000.

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

21 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.

22 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 für Informatik, 2018.

23 M. Szykuła and A. Zyzik. An Improved Algorithm for Finding the Shortest Synchronizing Words. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms (ESA 2022)*, volume 244 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 85:1–85:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

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

25 M. V. Volkov. Synchronization of finite automata. *Uspekhi Matematicheskikh Nauk*, 77:53–130, 2022. in Russian.