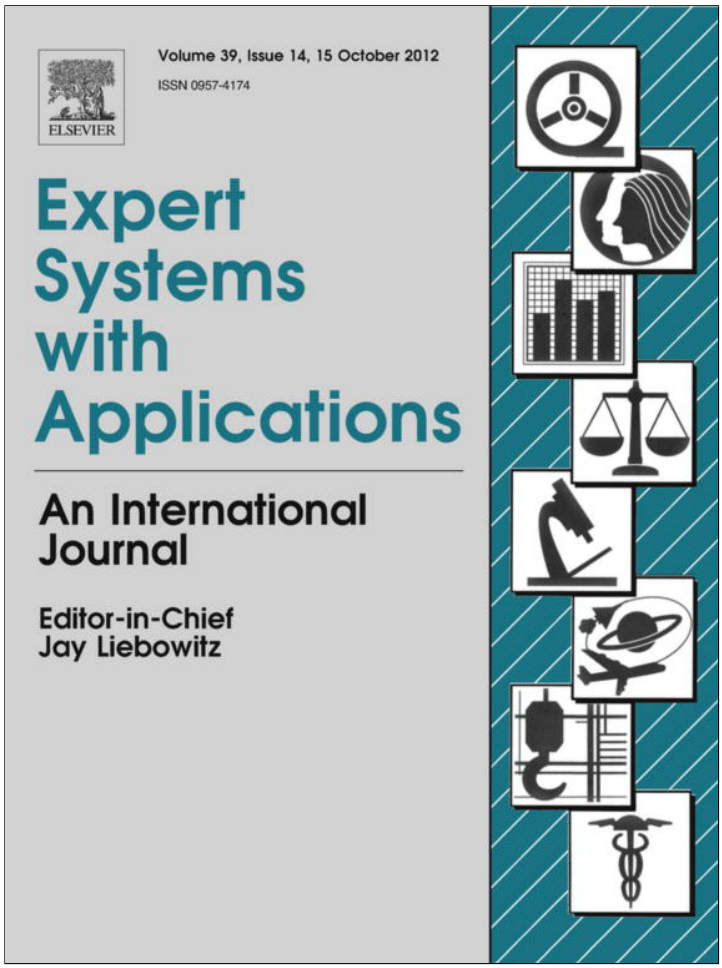

Expert Systems with Applications 39 (2012) 11746–11757

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Effective synchronizing algorithms

R. Kudłacik [a], A. Roman [b,*], H. Wagner [b]

[a] IBM Poland, SWG Krakow Laboratory, Armii Krajowej 18, 30-150 Krakow, Poland
[b] Institute of Computer Science, Jagiellonian University, Łojasiewicza 6, 30-348 Krakow, Poland

ARTICLE INFO

Keywords:
Circuit testing
Conformance testing
Synchronizing sequences
Synchronizing automata
Reset word
Synchronizing algorithm

ABSTRACT

The notion of a synchronizing sequence plays an important role in the model-based testing of reactive systems, such as sequential circuits or communication protocols. The main problem in this approach is to find the shortest possible sequence which synchronizes the automaton being a model of the system under test. This can be done with a synchronizing algorithm. In this paper we analyze the synchronizing algorithms described in the literature, both exact (with exponential runtime) and greedy (polynomial). We investigate the implementation of the exact algorithm and show how this implementation can be optimized by use of some efficient data structures. We also propose a new greedy algorithm, which relies on some new heuristics. We compare our algorithms with the existing ones, with respect to both runtime and quality aspect.

1. Introduction

Synchronizing words (called also: synchronizing sequences, reset sequences, reset words or recurrent words) play an important role in the model-based testing of reactive systems (Broy, Jonsson, Katoen, Leucker, & Pretschner, 2005). In presence, with advanced computer technology, systems are getting larger and more complicated, but also less reliable. Therefore, testing is indispensable part of system design and implementation. Finite automata are the most frequently used models that describe structure and behavior of the reactive systems, such as sequential circuits, certain types of programs, and, more recently, communication protocols (Fukada, Nakata, Kitamichi, Higashino, & Cavalli, 2001; Ponce, Csopaki, & Tarnay, 1994; Zhao, Liu, Guo, & Zhang, 2010). Because of its practical importance and theoretical interest, the problem of testing finite state machines has been studied in different areas and at various times. Originally, in 1950s and 1960s, the researchers work in this area was motivated by automata theory and sequential circuit testing. The area seemed to have mostly died down, but in 1990s the problem was resurrected due to its applications to conformance testing of communication protocols.

The problem of conformance testing can be described as follows (Lee & Yannakakis, 1996). Let there be given a finite state machine M_S which acts as the system specification and for which we know completely its internal structure. Let M_I be another machine, which is the alleged implementation of the system and for which we can only observe its behavior. We want to test whether M_I correctly implements or conforms to M_S.

Synchronizing words allow us to bring the machine into one state, no matter which state we currently are in. This helps much in designing effective test cases, e.g. for sequential circuits. In Pomeranz and Reddy (1998) authors show a class of faults for which a synchronizing word for the faulty circuit can be easily determined from the synchronizing word of the fault free circuit. They also consider circuits that have a reset mechanism, and show how reset can ensure that no single fault would cause the circuit to become unsynchronizable.

In Hyunwoo, Somenzi, and Pixley (1993) a framework and algorithms for test generation based on the multiple observation time strategy are developed by taking advantage of synchronizing words. When a circuit is synchronizable, test generation can employ the multiple observation time strategy and provide better fault coverage, while using the conventional tester operation model. The authors investigate how a synchronizing word simplifies test generation.

The central problem in the approach based on the synchronizing words is to find the shortest one for a given automaton. As the problem is $\mathcal{NP}$-hard (see Section 2), the polynomial algorithms cannot be optimal, that is they cannot find the *shortest* possible synchronizing words (unless $\mathcal{P} = \mathcal{NP}$, which is strongly believed to be false).

In last years some efforts were made in the field of algorithmic approach for finding short synchronizing words (Deshmukh & Hawat, 1994). Pixley, Jeong, and Hachtel (1994) presented an efficient method based upon the universal alignment theorem

* Corresponding author.
E-mail addresses: rafal.kudlacik@gmail.com (R. Kudłacik), roman@ii.uj.edu.pl (A. Roman), hub.wag@gmail.com (H. Wagner).

0957-4174/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.eswa.2012.04.079

and binary decision diagrams to compute a synchronizing word. There are also Natarajan (1986) and Eppstein (1990) algorithms.

The problem of synchronizing finite state automata has a long history. While its statement is simple (find a word that sends all states to one state), there are still some important questions to be answered. One of the most intriguing issues is the famous Černý Conjecture (Černý, Pirická, & Rosenauerová, 1971), which states that for any n-state synchronizing automaton there exists a synchronizing word of length at most $(n-1)^2$. Should the conjecture be true, this would be a strict upper bound, as there exist automata with minimal synchronizing words of length exactly $(n-1)^2$. The Černý Conjecture has profound theoretical significance (remaining one of the last 'basic' unanswered questions in the field of automata theory, especially after the Road Coloring Problem has been recently solved by Trahtman (2009)). On the other hand, there are several practical applications of finding short reset sequences: part orienters (Natarajan, 1986), finding one's location on a map/graph (Kari, 2002), resetting biocomputers (Ananichev & Volkov, 2003), networking (determining a leader in a network) (Kari, 2002) and testing electronic circuits, mentioned above. Clearly, finding short reset words is important both for theoretical and practical reasons.

The paper is organized as follows. In Section 2 we give the basic definitions on automata and synchronizing words. In Section 3 we introduce two auxiliary constructions which are commonly used in the synchronizing algorithms. In Section 4 we present the well-known synchronizing algorithms, both exact and greedy. In Sections 5 and 6 we present our two main results: application of efficient data structures to the exact synchronizing algorithm and a new, efficient heuristic algorithm. Both sections end with the experimental results and efficiency comparison to other algorithms.

2. Synchronizing words

An *alphabet* is a nonempty, finite set. A *word* over some alphabet A is a sequence of letters from A. The *length* of a word w is the number of its letters and is denoted by $|w|$. By ε we denote the *empty word* of length 0. If A is an alphabet, by A^* we denote the set all words over A. For example, if $A=\{a,b\}$, then $A^*=\{\varepsilon,a,b,aa,ab,ba,bb,aaa,\ldots\}$. The catenation of words is denoted by a dot: if u, $v\in A^*$, then $u\cdot v=uv$. A *finite state automaton* is a triple $\mathcal{A}=(Q,A,\delta)$, where Q is a finite set of states, A is an alphabet and δ is a transition function, $\delta: Q\times A\to Q$. Note that initial and terminal states are not marked – we are not interested in languages accepted by automata, but rather in automaton action itself. In the following, for the sake of simplicity, we will use the word 'automaton' instead of 'a finite state automaton'. The transition function can be extended to $\mathcal{P}(Q)\times A^*$, that is, to the sets of states and words over A. The same symbol δ will be used to refer to the extended function $\delta:\mathcal{P}(Q)\times A^*\to\mathcal{P}(Q)$. This makes no confusion: $\forall P\subseteq Q, a\in A, w\in A^*$

$$\delta(P,\varepsilon)=P,$$
$$\delta(P,aw)=\bigcup_{p\in P}\{\delta(\delta(p,a),w)\}.$$

A word w is called a *synchronizing word* for $\mathcal{A}=(Q,A,\delta)$ iff $|\delta(Q,w)|=1$. We say that such a word *synchronizes* $\mathcal{A}$. We also say that $\mathcal{A}=(Q,A,\delta)$ is *synchronizing* if there exists $w\in A^*$ that synchronizes it. If, for a given $\mathcal{A}$, there is no shorter synchronizing word than w, the word w is called *the shortest synchronizing word* (SSW) for $\mathcal{A}$. There are two main algorithmic problems in the synchronization theory: in the first one, given a synchronizing automaton $\mathcal{A}=(Q,A,\delta)$ we ask about SSW for $\mathcal{A}$. In the second one we ask to find any synchronizing word, not necessarily the shortest (but, of course, the shortest word is found, the better), in a reasonable time. These problems can be restated in the form of the following decision problems:

Problem FIND-SSW.

Input: a synchronizing automaton $\mathcal{A}$ and $k\in\mathbb{N}$.
Output: YES iff the shortest word synchronizing $\mathcal{A}$ has length k.

Problem FIND-SW-OF-LENGTH-K

Input: a synchronizing automaton $\mathcal{A}$ and $k\in\mathbb{N}$.
Output: YES iff there exists a synchronizing word of length k for $\mathcal{A}$.

The decision problem FIND-MSW has been recently shown to be $\mathcal{DP}$-complete (Olschewski & Ummels, 2010). The decision problem FIND-SW-OF-LENGTH-K is $\mathcal{NP}$-complete (Eppstein, 1990). It is well-known that the length of SSW for an n-state synchronizing automaton is at most $\frac{n^3-n}{6}$ (Klyachko, Rystsov, & Spivak, 1987; Pin, 1983). The Černý conjecture states that this length can be bounded by $(n-1)^2$. Černý showed (Černý, 1964) that for each $n\geqslant 1$ there exists an automaton with SSW of length $(n-1)^2$, so the conjectured bound is tight. These automata are called the Černý automata. An n-state Černy automaton will be denoted by $\mathcal{C}_n$. Černy automaton is defined over a two-element alphabet $A=\{a,b\}$ and its transition function is as follows:

$$\forall q\in\{0,\ldots,n-1\}\ \delta(q,x)=\begin{cases}(q+1)\bmod n & \text{if } x=a\\ q & \text{if } x=b\wedge q\neq n-1\\ 0 & \text{if } x=b\wedge q=n-1\end{cases}\tag{1}$$

Černý automata are very important, as automata with $|SSW|=(n-1)^2$ are very rare. Only eight such automata are known that are not isomorphic with the Černy automata (Trahtman, 2006).

3. Auxiliary constructions

In this section we describe two auxiliary constructions used throughout the paper. Let $\mathcal{A}=(Q,A,\delta)$ be a synchronizing automaton. A *pair automaton* for $\mathcal{A}$ is the automaton $\mathcal{A}^2=(Q',A,\delta')$, where:

$$Q'=\bigcup_{p,q\in Q\wedge p\neq q}\{\{p,q\}\}\cup\{0\},$$
$$\delta':Q'\times A\to Q',$$
$$\delta'(\{p,q\},l)=\begin{cases}\{\delta(p,l),\delta(q,l)\} & \text{if } \delta(p,l)\neq\delta(q,l),\\ 0 & \text{otherwise,}\end{cases}$$
$$\delta'(0,l)=0\ \forall l\in A.$$

Let $\mathcal{A}=(Q,A,\delta)$ be an automat on. A sequence $(q_1,q_2),(q_2,q_3),\ldots,(q_l,q_{l+1}),q_1,\ldots,q_{l+1}\in Q$, is called a *path* in $\mathcal{A}$, if for each $i=1,\ldots,l$ there exists $a_i\in A$, such that $\delta(q_i,a_i)=q_{i+1}$. We will identify such a path with a word $a_1a_2\ldots a_l$ (notice that if there is more than one letter transforming some q_i into q_{i+1}, then the path including (q_i,q_{i+1}) can be identified with more than one word). Pair automaton shows how the pairs of states behave when words are applied to the original automaton. If $p,q\in S\subseteq Q$ and w is a path leading from $\{p,q\}$ to 0, it means that $|\delta(S,w)|<|S|$, where $p,q\in S$. In such a situation we say that pair $\{p,q\}$ of states was *synchronized* by w. Pair

automaton is utilized in all heuristic algorithms, see Sections 4.3, 4.4 and 4.5. The next proposition is a straightforward, but very important fact, utilized in all heuristic algorithms.

Proposition 1. *A word $w \in A^*$ synchronizes $\mathcal{A}^2$ iff w synchronizes $\mathcal{A}$.*

Proposition 1 implies the following necessary and sufficient condition for $\mathcal{A}$ to be synchronized:

Proposition 2. *$\mathcal{A}$ is synchronizing iff each pair of its states is synchronizing.*

The problem of finding SSW can be restated as a problem of path-searching in a so-called power-set automaton (or power automaton for short) of $\mathcal{A}$. A *power-set automaton* for $\mathcal{A} = (Q, A, \delta)$ is an automaton $\mathcal{P}(\mathcal{A}) = (2^Q, A, \Delta)$, where:

$$2^Q = \{P \subset Q\} \setminus \{\emptyset\},$$
$$\Delta : 2^Q \times A \to 2^Q,$$
$$\Delta(q, l) = \bigcup_{s \in q} \{\delta(s, l)\} \ \forall q \in 2^Q, l \in A.$$

Like for δ, we can extend Δ to $2^Q \times A^*$. Let $\mathcal{P}(\mathcal{A}) = (2^Q, A, \Delta)$ be a power automaton of $\mathcal{A} = (Q, A, \delta)$. State $Q \in 2^Q$ will be called a *start state* of $\mathcal{P}(\mathcal{A})$. The size of the power automaton is exponential in the size of the original automaton. There are $2^{|Q|} - 1$ states and $|A|(2^{|Q|} - 1)$ edges. States of the power automaton represent subsets of states of the input automaton and are labeled by the corresponding subsets. We will only consider a subautomaton of the power automaton which is reachable from the start state. Specifically, when we say that the power automaton is small, we mean that the reachable subautomaton is small. Černy automata $\mathcal{C}_n$ are the interesting examples here, as all states of their power automata are reachable from the start state. We will sometimes refer to the "size of state s". This means that s is a state of the power automaton and it represents an $|s|$-element subset s of Q.

Since edges in pair automaton represent transitions between subsets of states, the power automaton can be thought of as a way of expressing the global behavior of the input automaton when certain letter (or word) is applied. In contrast, pair-automaton describes local behavior only (it shows how the pairs of states are transformed).

Proposition 3. *The sets of synchronizing words in $\mathcal{A}$ and $\mathcal{P}(\mathcal{A})$ coincide. It means that $\mathcal{A}$ is synchronizing iff $\mathcal{P}(\mathcal{A})$ is synchronizing.*

It is clear that in a power automaton a path leading from $Q \in 2^Q$ to any state $F \in 2^Q$, such that $|F| = 1$, represents a synchronizing word for $\mathcal{A}$. Also, the shortest such path determines the shortest synchronizing word for $\mathcal{A}$. So the entire problem can be rephrased as a basic graph problem. This is convenient as the single-source path-searching algorithms (exact or otherwise) have been extensively studied. Also, augmenting the generic path-searching methods with knowledge specific to this problem may give some interesting results.

4. Synchronizing algorithms

In this section we describe 5 synchronizing algorithms, that is, the algorithms that find a synchronizing word for a given automaton. Two of them (EXACT and SEMIGROUP) are exponential ones that always find the shortest synchronizing words. Three others (NATARAJAN, EPPSTEIN and SYNCHROP) are heuristic algorithms working in polynomial time, so they are faster, but they find not necessarily the shortest synchronizing words. In the following sections we assume $|Q| = n$.

4.1. Exact exponential algorithm

There are two well-known algorithms finding the shortest synchronizing words. Due to the fact that this problem is $\mathcal{NP}$-hard, their runtime complexity is exponential in the size of the input automaton, which limits their use.

The standard exact algorithm is a simple breadth-first-search in the power automaton. The runtime is $\Omega(2^n)$ in the worst case. Standard implementation requires $\Omega(2^n n)$ space. Due to these discouraging fact this algorithm is often disregarded in the literature.

Algorithm 1 EXACT ALGORITHM($\mathcal{A}$)

```
1: Input: an automaton A = (Q, A, δ)
2: Output: SSW of A (if exists)
3: queue Q ← empty
4: push Q into queue Q
5: mark Q as visited
6: while Q is not empty
7:   S ← pop(Q)
8:   foreach a ∈ A
9:     T ← δ(S, a)
10:    if |T| = 1
11:      return reversed path from T to Q
12:    if T is not visited
13:      push T into Q
14:      mark T as visited
15: return A is not synchronizing
```

4.2. Semigroup algorithm

Another algorithm (which is typically more memory-efficient) was described in Trahtman (2006) and uses a notion of syntactic semigroup. Let $\mathcal{A} = (Q, A, \delta)$ be an automaton. Alphabet letters (and also words over A) represent functions $Q \to Q$, so if f is a function from Q to Q and $w \in A^*$, by $f \circ w$ we denote the composition of two functions: f and a function corresponding to w. Syntactic semigroup for $\mathcal{A}$ is constructed as follows: process all words over A^* in the lexicographic order. If a processed word defines a new function f: $Q \to Q$, add f to list L. The procedure is stopped in two cases: (1) when $\forall f \in L \forall a \in A f \circ a \in L$, that is, when no new function can be defined; (2) when a constant function (mapping all elements into one element) is found. The word corresponding to the constant function is SSW, as words were processed in the lexicographic order. The semigroup algorithm does not require a costly power automaton construction phase, but its standard implementation is terribly inefficient in the worst case. So it is slightly better than the power automaton algorithm, but the above fact limits its use only to small automata.

The algorithm's runtime complexity is $O(|A| n \cdot s^2)$ with $O(n \cdot s)$ space required (Trahtman, 2006), where s is the size of the syntactic semigroup S. Syntactic semigroup size can be as big as n^n, but since only a subset of S (containing only words no longer than SSW) is considered, the average runtime is usually much lower.

The semigroup algorithm is used in a well-known synchronization package TESTAS. Its worst-time complexity can be drastically reduced and we show it in Section 5.

4.3. Natarajan algorithm

One of the first heuristic algorithms for finding short synchronizing words was provided by Natarajan (1986). The algorithm is shown in Listing 2.

Algorithm 2 NATARAJAN($\mathcal{A}$)

1: **Input**: synchronizing automaton $\mathcal{A} = (Q, A, \delta)$
2: **Output**: synchronizing word for $\mathcal{A}$
3: $Q \leftarrow \{1,2,\ldots,n\}$; $s \leftarrow \varepsilon$
4: **while** $|Q| > 1$:
5: choose two states $p, q \in Q$
6: $w \leftarrow$ the shortest path from $\{p,q\}$ to 0
7: $Q \leftarrow \delta(Q,w)$
8: $s \leftarrow s.w$
9: **return** s

A loop in line 4. is performed $O(n)$ times. The shortest path (line 6.) can be found in $O(|A|n^2)$. Transformation in line 7. is done in $O(n^3)$, because $|Q| = O(n)$ and $|w| = O(n^2)$. Hence, the total complexity is $O(|A|n^3 + n^4)$.

4.4. Eppstein and cycle algorithms

Eppstein proposed a modification of Natarajan's algorithm. The modification is based on a preprocessing in which for each pair of states we compute the first letter of the shortest word synchronizing these states. Eppstein has shown that this preprocessing allows us to reduce the complexity to $O(n^3 + |A|n^2)$.

CYCLE is a slight modification of EPPSTEIN. In CYCLE, when a pair of states in synchronized into some state q, it is required that in the next step q must be one of the elements in the chosen pair. CYCLE works optimally for Černý automata, that is, always returns SSW.

4.5. SYNCHROP algorithm

SYNCHROP (and its modified version, SYNCHROPL) algorithm (Roman, 2009) is, in comparison to NATARAJAN, a 'one-step-ahead' procedure – we do not choose arbitrary pair of states as in line 5. of NATARAJAN. Let $w(p,q)$ be the shortest word synchronizing $\{p,q\}$. For each $\{p,q\}$ we check how the set of states in the pair automaton will be transformed if we apply $w(p,q)$ to all states we currently are in. Each transformation is rated in terms of some heuristically defined cost function. We choose the pair with the lowest cost function. The remaining part of the algorithm is exactly the same as in NATARAJAN. In its original version, SYNCHROP algorithm does not use the preprocessing introduced in EPPSTEIN. Therefore, its complexity is $O(n^5 + |A|n^2)$. The detailed description and discussion on SYNCHROP properties and complexity is given in Section 6.

5. Optimizing exponential algorithms

In this section we deal with the exact synchronizing algorithms. We show how the selection of the efficient data structures affects on the time complexity. Let us consider the basic version of the algorithm shown in Listing 1. While the algorithm looks very simple, its performance greatly depends on the data-structures used. The following aspects of the algorithm must be considered:

1. transition function computation,
2. state representation,
3. queue implementation,
4. visited states' set implementation,
5. predecessor tree implementation (required, if the actual SSW must be returned rather than its length).

Judging by the complexity (given in Trahtman (2006)) of the SEMIGROUP algorithm implementation in TESTAS, checking if t was previously visited (Listing 1, line 12.) is assumed to be performed in $\Theta(nm)$, where m is the number of elements visited. This step can easily be done in time $n\log m$ using the standard tree-based dictionaries. So the worst case runtime complexity can easily be reduced from $\Theta(|A|ns^2)$ to $\Theta(|A|ns\log s)$.

Another simple optimization is possible. Note that the original algorithm generates sequences of states of size n (and not sets). We can treat these elements as sets without losing any valuable information. This should also speed the process up: semigroup size s can reach the value of n^n (and reaches 2^{2n} for $\mathcal{C}_n$), while there are at most 2^n subsets of the considered set.

Finally, for small n, a trick can be used: sets can be mapped to integers. This technique will be described in more details later. An ordinary array can be used to check if a set was previously added to the visited sets (actually, similar trick can be applied when sequences of size n are considered: radix n rather than 2 must be used). This allows us to skip the logarithmic part in $\Theta(|A|ns\log s)$ yielding $\Theta(|A|ns)$ (assuming that n is small). Of course, one could argue that there is no point in analyzing asymptotic complexity with bounded values of input size. In such cases this notation should be understood as a way to express the order of complexity.

This makes a really big difference. For example, TESTAS (which apparently uses this algorithm) cannot handle calculating SSW of $\mathcal{C}_n$ for $n \geqslant 16$. After these simple optimizations, all automata of size up to 26 (or more) should be handled easily (space complexity becomes a bigger problem in case of slowly-synchronizing automata). More detailed comparison will be shown later.

Interestingly, after performing these optimizations, the algorithm is very similar to the power-automaton-based approach. In fact, we will try to merge the best features of these two approaches.

5.1. Power automaton traversal

We would like to focus now on the algorithm that utilizes the concept of the power-automaton: a breadth-first-search is performed, beginning with the start state (set of all states of the input automaton). When a singleton state is found, the SSW has been found and the computation can be terminated. In the effect, typically only a subset of states of the power automaton is utilized. This is an important fact that will enable us to examine larger automata.

5.1.1. On-line transition function computation

The power automaton transition function can be calculated online (*i.e.* whenever it is required). In order to compute a single power automaton transition, one to n transition functions of the original automaton must be calculated. This is reasonable and is, in fact, a standard approach used. Typically the resulting runtime complexity is $O(|A|n2^nM(n))$, where $M(n)$ is the cost of performing one mapping of state into an associated value. For small n it can be assumed to be $O(|A|n2^n)$ (this is a slight abuse of the O notation, as this algorithm is limited to small n; in fact we are not interested in asymptotic behavior of this function).

5.1.2. Off-line transition function computation

It is possible to generate all transitions in the power automaton in an amortized constant time. This approach leads us to $\Theta(|A|2^n)$ complexity of calculating SSW. Currently, memory requirements make it usable only for $n < 30$, so we will only investigate this case. Power automaton's states $S_1 \ldots S_{2^n-1}$ are generated in Gray's code order, which can be done in amortized constant time. This way, consecutive states differ on exactly one position (i.e $|S_i \oplus S_{i+1}| = 1$, where $\oplus$ denotes a symmetric-difference operation).

Power automaton transition for certain $l \in A$ can be expressed as

$$\Delta(S_i, l) = \delta(S_i \cap S_{i-1}, l) \cup \Delta(S_i \oplus S_{i-1}, l), \tag{2}$$

where

$$|S_i \cap S_{i-1}| = |S_i| - 1,$$
$$|S_i \oplus S_{i-1}| = 1.$$

If S_i is the ith generated state, $\delta(S_i \cap S_{i-1}, l)$ can be determined in constant time. This can be done by storing the count of each element of the current transition's function value. This information can be updated in constant time.

Some additional pre-computation is necessary, so that logarithm of a number can be computed in constant time. This is required to map a power automaton's state (representing the symmetric difference of consecutive states) into input automaton's state. Since $|S_i \oplus S_{i-1}| = 1, S_i \oplus S_{i-1}$ is encoded as an integer I being a power of two. So $\log(I)$ denotes the state's number. Listing 3 shows how to calculate this value quickly. Finally, the entire algorithm shown in Listing 4 can be run in a constant time.

Algorithm 3 FAST_LOG2(n)

```
1: Input: integer n = 2^k
2: Output: log2 n = k
3: r ← right (n)// right half, r < 2^16
4: if r > 0:
5:   return log2[r]// precomputed value
6: l ← left (n) shr 16
7: return 16 + log2[l]// assuming 32 bit integers are used
```

Algorithm 4 FAST POWER AUTOMATON TRANSITION (S_i,S_{i+1}, prev_trans, image_count, l)

```
1: Input: States S_i, S_{i+1}; transition function for S_i; count of each
   element in the transition function's image; letter l ∈ A for
   which the transition function is calculated
2: Output: power automaton transition for a given state and
   letter
3: ex ← S_i XOR S_{i+1}// bitwise symmetric difference
4: in ← S_i AND next// bitwise and
5: ret ← prev_trans
6: change ← fast_log2(ex)// changed state's number
7: change_to ← delta (change, l)
8: if in == prev:// change was added
9:    image_count[change]+ = 1
10:   ret = ret OR change_to
11: else// change was removed
12:   image_count[change_to]− = 1
13: if image_count[change_to] == 0
14:   ret  = ret XOR change_to
15: return ret
```

A full graph representing the power automaton is constructed in memory.

This approach is good for calculating SSW of slowly-synchronizing automata. It takes exactly $\Theta(|A|2^n)$ integer operations (that is, never takes less, unlike other implementations). It is not suitable for larger ($n > 30$) automata due to memory requirements. It is also not recommended for random and fast-synchronizing automata as they tend to have small power automata. This should be a good choice when the inexact algorithm (run prior to the exact one) shows that SSW may be long.

5.1.3. Mapping states to arbitrary information

In the algorithm a set of visited states must be maintained. We will consider a more general solution: mapping states into arbitrary values (when a Boolean value is used, this approach is equivalent to defining a subset by its characteristic function). This method will be useful for storing predecessor tree.

5.1.4. Dense mapping

When we can afford keeping the entire mapping in memory, the situation is rather simple. To each set can be assigned an integer that will uniquely identify this set. There exists a convenient correspondence σ:

$$\sigma : 2^{\{1,\ldots,n\}} \to \mathbb{N},$$
$$\sigma^{-1} : \{0,\ldots,2^n - 1\} \to 2^{\{1,\ldots,n\}},$$
$$\sigma(S) = \sum_{i \in S} 2^{i-1},$$
$$\sigma^{-1}(I) = \{i \leqslant n : \text{bit } (i-1) \text{ of } I \text{ is set to } 1\}.$$

In other words, subsets of a fixed set can be represented as its characteristic vector. This binary vector can be encoded as an integer. It is a common technique. All relevant operations can be performed quickly. Hence, an ordinary array (of size 2^n) can be used to map power automaton's states into arbitrary data (for example, an information if a given set was visited earlier during the search). Since the array must fit into memory, n must be small.

5.1.5. Sparse mapping

In case where only a subset (of unknown size) must be mapped, sparse data structures should be used. Such structures include tree-based dictionaries (like various BSTs) and hash tables. Trees require a strict weak ordering on elements (that is a proper '<' predicate must be supplied). Note that $x = y \iff \neg(x < y) \wedge \neg(y < x)$.

Hash tables require an equality predicate (=). Also, it is required that hash value h can be calculated for each element.

Tree-based dictionaries typically guarantee that insertions and retrievals perform $\Theta(log_2 m)$ key comparisons, where m is the number of stored keys. Hence, the complexity of performing key comparisons must be included in the estimation of the total complexity (this fact seems to be often omitted).

Hash tables promise insertions and retrievals in $O(1)$ time on average. Once more, the complexity of comparing keys and calculating hashes must be taken into account.

Using a trie data structure is also an option, but no efficient implementations seem to be available. An optimized implementation based on a compact two-array approach would be suitable for our needs.

There exist standard implementations of tree-based and hash-based dictionaries and we will not delve into further details here. We used `std::map` from STL library (which implements red–black trees) and `boost::unordered_map` from the boost library (Björn, 2005).

5.1.6. State-set representation for the sparse mapping

Since only a small subautomaton of the power automaton must typically be stored, n can be much bigger (>80). In effect, power automaton's states can no longer be encoded into a single integer value. An alternative representation must be devised. Essentially, we need to represent subsets of $\{1,2,\ldots,n\}$. We will now investigate various data-structures that can be used for this purpose.

Let a data structure $\mathcal{S}$ represent a set $S \subseteq \{1,2,\ldots,n\}$. By iteration over $\mathcal{S}$ we mean enumerating elements of S, preferably in sorted order. So if $\mathcal{S}$ represents the set $\{1,2,3\}$, iterating over $\mathcal{S}$ yields elements 1, 2, 3, and signalizes that no more elements are present.

5.1.7. Tree-based sets

Using the standard set structures based on AVL or red–black trees (like `std::set` from the C++standard library) as state representation severely decreases performance. Also, memory footprint is unacceptable.

5.1.8. Sorted arrays

Arrays of fixed size n (or dynamically sized vectors) can be used as long as the sorted order (or any other canonical order that enables consistent comparisons required by the dictionary data-structures) is preserved. This solution also has some drawbacks, as comparing two states requires comparing as many as n integers. Furthermore, each power automaton transition requires sorting the resulting set (array) of states. Also taking $h(state)$ requires combining hashes of all values ($n/2$ on average).

5.1.9. Fast bit-vector

It turns out that an efficient, yet simple, data structure can be used to store individual states of power-automaton. It will be a mix of three described approaches. It encodes states into several integer values (using a modification of the described technique). Contained states can be iterated in sorted order (like in case of `std::set`) that is preserved with no additional overhead (like explicit sorting in case of using arrays). Finally, it uses a plain array to store the values.

The most important improvement is that efficient comparisons and hash calculations can be performed. Normally, individual values of the contained sets are compared. Here the data-structure is treated as a sequence of bits (or integers). This way, the actual elements do not have to be decoded. The raw integers representing subsets of values are compared. This is a vast improvement, since as many as 32 (or 64 depending on the CPU) values are compared in parallel. Hashes are calculated similarly.

This structure is also much more compact: only 1 bit per state is needed. An array requires at least 8 bits (typically 16). The memory overhead of the standard set data-structures is much greater (at least 70 bits). As an improvement for very sparse sets, iteration can be modified: rather than checking bits one by one, bit-masks of certain size (16,8) can be applied to cull large empty chunks. This is an extension of a basic approach that skips empty (zero) blocks. The only problem with this solution is that the iteration requires amortized $\Theta(n)$ time in the worst case (typically amortized time is constant). Creation of the data-structure takes $n/32$ time (memory must be cleared), so improving this element might be beneficial.

This solution can be extended by storing an additional unsorted array of values. It would speed the iteration process up, but memory consumption would be increased by an unacceptable factor (each stored value would require at least 9 bits instead of 1).

5.1.10. Bit-vector indexed with an implicit binary tree

Iteration speed can be increased, only slightly affecting the memory footprint. Exploiting the fact that a fixed universe of keys is considered, an index (represented by an implicit tree) can be built over a bit-vector. The main advantage over standard (explicit) trees is that the pointers to children are implicit (that is, they are not stored in memory). It can be implemented similarly to a binary-heap: an array (or bit-vector) A is used, elements $A[2i]$ and $A[2i+1]$ represent children of node i (nodes are numbered from 1). Node $A\left[\lfloor \frac{i}{2} \rfloor\right]$ is the parent of i and $A\left[\lfloor \frac{i+1}{2} \rfloor\right]$ is the parent of the right-sibling (actually this represents sibling's or grand-sibling's parent; the important thing is that $i+1$ is on the same level as i (unless i is the rightmost node)).

Each node contains one bit of information, answering the following question: is my sub-tree empty? Specifically, if the root node's sub-tree is empty, it means the whole tree is empty. If leaf p is non-empty, it means that the value $p-n$ is present in the set. Note that 32 (or 64) nodes can be encoded in one integer.

For simplicity's sake we will assume that n is a power of 2. There are $\log(n)+1$ levels in the tree. Number of nodes at level L equals 2^L, so the total number of nodes is $2n$. Note that this structure can be accessed just as a plain bit-vector (values $A[n]\ldots A[2n-1]$ directly represent contained elements). Hence, checking if $x \in S$ takes $O(1)$ time.

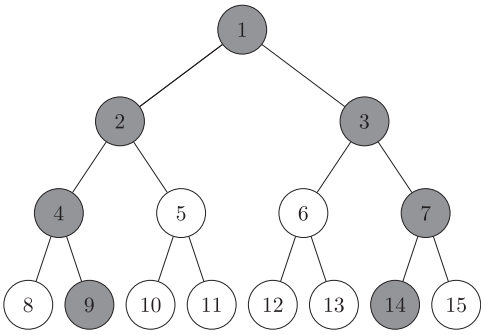

Fig. 1. Exemplary bit-vector indexed with a tree.

Fig. 1 shows the structure for $n=8$. This structure can represent subsets of $\{0,\ldots,n-1\}$. The figure shows the state of the structure after inserting elements 1 and 6. Gray color represents marked nodes (that is, nodes with value 1). Note that respective nodes 9 and 14 are gray. Also, the paths from these nodes to root are gray. Nodes 8,...,15 directly represent the elements in a current set.

Adding an element (we are not interested in removal in our applications, so this operation will not be described) requires updating all ancestors up to the root. Adding an element to an empty structure takes $\log n$ time. Next time an element is added, when a non-empty node is found, a path from this node to the root is already updated, so the overhead is smaller. It seems that the cost of populating the structure with n distinct values is $n\log n$. Noticing that each node (including internal nodes) will be marked as non-empty only once, yields a more precise result, $2n$.

Let us investigate a more general case: any number of elements is added. Clearly, the amortized cost of adding new elements is non-decreasing in the number of elements in the set. We have already shown that when all elements are added, the cost is linear. This value cannot be greater for a smaller number of elements, since the cost must be positive. The amortized complexity of inserting one element is logarithmic in the worst case, but the total complexity (that is, the sum of costs of all operations) is never worse than linear. Of course, this is nothing new – the same thing can be achieved with an ordinary bit-vector.

The complexity of iteration is more interesting. Iteration uses the function shown in Listing 5.

Algorithm 5 UPPERBOUND(tree, n, elem)

```
1: Input: n-element set, its subset represented by tree and an
   element elem from the set
2: Output: smallest element next present in the set such that
   next ⩾ elem
3: node ← n + elem
4: if (tree[node])
5:   return elem
6: prev_node ← node
7: while ((not tree[node]) and (node != root)) 8:
   prev_node ← node
9:    node ← parent (node)
10: return down (tree, n, prev_node, node)
```

Algorithm 6 DOWN(tree, n, from, node)

```
1: if (leaf (node) and (not tree[node]))
2:   return node-n// found
3: if (leaf (node) and tree[node])
4:   return-∞// found nothing
5: if tree[left (node)] and left (node) > from
6:   return down (tree, n, node, left (node))
7: return down (tree, n, node, right (node))
```

The UPPERBOUND goes up to the root searching for the non-empty nodes. When a node is found, we go down, following the leftmost path (leading to leaves with indices greater than the start node) of non-empty nodes. Clearly, at most $2\log n$ nodes are visited. This happens, for example, when only one value is present. The entire structure can be iterated using the function ITERATE (see Listing 7).

Algorithm 7 ITERATE (tree, n)

```
1: Input: structure S
2: Output: next elements of S
3: elem = −1
4: while(elem != −∞)
5:     elem = UPPERBOUND(tree, n, elem + 1)
1:     yield elem // the next element was found
```

Let us investigate the amortized complexity of iterating over a full set. This is trivial, since UPPERBOUND function returns immediately in line 5. Hence, the amortized complexity is constant when all elements are present. We already mentioned that when one element is present, the amortized complexity (as well as total) is $2\log n$.

Instead of going up to a parent we can visit our grand-sibling's parent without skipping any valuable nodes (sometimes we could visit the grand-sibling itself if it is non-empty, but this happens only once per UPPERBOUND call and lowers performance in the worst case). The remaining part of the algorithm is unchanged: if the current node is non-empty (marked), we start moving down. Otherwise, we continue moving right-up. This small change is important, as it enables us to perform more detailed complexity analysis. Note that the cost (counted in the number of visited nodes) between two leaves p, q is bounded by $3\log(|p-q|)$. The search process can be divided into two phases. The search starts from node p.

1. Right-up jumps are performed until a non-empty node is found. Successive right-up jumps cull sub-trees of exponentially growing sizes: 1, 2, 4 etc. Since there are $|p-q|$ empty leaves between the considered leaves, it is enough to perform $\log|p-q|$ jumps.
2. Now, the left-most non-empty path is followed. For each performed up or right-up jump a down-move must be performed (there were at most $\log|p-q|$ such jumps). Each down-move requires checking two nodes.

It can be seen that at most $3\log(|p-q|)$ nodes will be accessed.

Let us now investigate the worst case. Assuming there are at least two elements (and one of them equals 0 for simplicity), they are iterated left-to-right: y_i is the number of elements checked during step i (that is, during the ith call to UPPERBOUND). Let us put $x_i = y_{i+1} - y_i \forall i < n$. Now, the cost of iterating through all elements equals $\sum_i x_i$. Note that $\sum_i y_i \leqslant n$, but we will assume $\sum_i y_i = n$, which is the worst case.

The total cost is $max_x(\sum_i \log(x_i)) = max_x(\log(\Pi_i x_i))$. From logarithm's monotonicity we only have to maximize the product, which happens when $x_i = x_j \forall i,j$. This leads to function $a^{n/a}$, where a is a parameter. Differentiating shows that it is maximized for $a = e$. Taking into account that we operate on positive integer values, we finally obtain that the cost is bounded by $O(\log\left(3^{\frac{n}{3}}\right)) = O(\frac{n}{3}\log 3) = O(n)$.

The worst case amortized complexity has been improved from $\Theta(n)$ to $\Theta(\log n)$, while worst case total complexity is still $O(n)$. Memory consumption is increased twice (two bits per value are required).

5.1.11. Other data-structures

Van Emde Boas Tree is an extension of the indexing tree concept described above. This heavy, recursive data-structure (each node contains a smaller VEB-Tree that is used as an index) enables iteration of s in $\Theta(|s|\log\log n)$, so it guarantees $\Theta(\log\log n)$ amortized complexity. This structure is complex, so it would be an improvement only for much bigger n. It is of no use in our applications.

5.1.12. Partial power-set automaton

The following technique can be used to reduce the amount of computations involved in calculating power automaton's transitions. This, as far as we know, novel technique is most useful for small n.

Let us define a *partial power automaton* of $\mathcal{A}$ for $X \subset Q$ as $\mathcal{P}_X(\mathcal{A}) = (2^Q, A, \Delta|_X)$. In other words, the transition function domain is restricted to X. Note that the co-domain does not change.

It is obvious that

$$S \subseteq 2^Q \wedge \bigcup S = Q \Rightarrow \mathcal{P}(\mathcal{A}) = \bigcup_{x \in S} \mathcal{P}_x(\mathcal{A}).$$

From a mathematical standpoint it is a trivial tautology, but it turns out that it can be useful for our purposes. Let us consider a simple case. Put $Q = \{1,\ldots,32\}$, $X = \{1,\ldots,16\}$, $Y = \{17,\ldots,32\}$. Then $|\mathcal{P}_X(\mathcal{A})| = |\mathcal{P}_Y(\mathcal{A})| = 2^{16} - 1$. Clearly, values of Δ_X and Δ_Y can be precomputed in $\approx 32 \cdot 2^8$ ($\approx n\sqrt{2^n}$ in general). Later, they can be used to construct the transition function of the entire power automaton by taking a union of transitions of the partial power automaton. Assuming that the union operation is faster then performing the transition in the original automaton, a speedup will occur. This is a low-level optimization, but experiments show that it can boost overall performance by a factor of 10.

It is indeed possible to perform a fast set-union operation. When the represented universe is fixed (in this case: {1, ...,32}), a set can be represented as its characteristic vector, encoded by an integer. Set-union is done by bit-wise OR operation, so the union of two states of size 32 is performed in one CPU instruction.

This approach can be generalized. Let m (dividing n, for simplicity) be the maximum value such that $m2^{n/m}$ transitions can be precomputed. Let us assume that our machine's word size is 32 bits. There are $\lceil\frac{n}{32}\rceil$ words necessary to store one transition value. We need m values, so we need to bit-or m values, which takes $\frac{mn}{32}$ operations. Normally we would need to perform exactly n transitions in the original automaton. For $m < 32$ this technique should be faster.

When $m = 2, n = 32$, exactly two results must be united. This can be done using one bit-or operation. In the result three integer operations must be performed (instead of $n/2$ on average). The preprocessing phase takes $\approx 2 \cdot 2^{16}$ operations.

5.2. Possible further improvements

As it was noted in Volkov (2008), an average length of SSW should be quite low (comparing to the conjectured $(n-1)^2$). According to the considerations of Volkov (referring to the paper

of Higgins (1988)), the expected SSW length is $O(n)$. More precisely, it can be proved that a randomly chosen n-state automaton with a sufficiently large alphabet is synchronizing with probability 1 as n goes to infinity and the length of its SSW does not exceed $2n$. Therefore, statistically, only a small part of the power automaton must be visited in the search process. Recently, Skvortsov and Tipikin (2011) provided experimentally the average length of SSW for a random n-state automaton over binary alphabet – it is $O(n^{0.55})$.

Regarding to the above facts, the described algorithm is not only exact, but also turns out to be quite efficient in an average case. Also, for small n, such an algorithm can be more effective than some polynomial-time solutions.

One more improvement can be introduced. It aims at reducing the runtime complexity for certain types of automata (such as Černý automaton) whose SSWs are of the form $w^k v, w, v \in A^*$. A similar heuristics was described in Trahtman (2006) to enhance the greedy algorithm. For each word wa such that $|\delta(Q,w)| > |\delta(Q,wa)|$, $w \in A^*$, $a \in A$, the powers of this word are checked. It may turn out that $(wa)^k$ is a synchronizing word (just as in case of the Černý automaton). Actually, many slowly synchronizing automata (*i.e.* with long SSW) fall into this category.

Note that by using this heuristics only a small fraction of the power automaton is visited, even though the length of SSW is quadratic and (normally) vast number of states would have to be traversed. Experimental data show that employing this heuristics for the Černý automata reduces the number of visited states to $\Theta(1.5^n) \approx \Theta(\sqrt{2^n})$.

5.3. Performance comparison

Table 1 shows how the described optimizations affected performance. The results are compared with the popular synchronization tool, TESTAS (v. 1.0). Fig. 2 shows the results graphically.

The measurements clearly show that the optimizations are very effective. Version 4 (using raw arrays of integers and partial power automaton concept), while significantly faster, cannot be used for larger automata. Other solutions, while slower, scale better and can be used for much larger automata, as long as the reachable power automaton size fits in memory.

Algorithm 3 (using hash tables and the described fast bit-vector) should be used for medium size automata. It can be more effective than algorithm 4 for smaller automata if the power automaton is relatively small. Approach 4 is superior for small ($n < 30$), slowly-synchronizing automata.

Note that all of these approaches are faster than the algorithm used in TESTAS, due to the described complexity improvements. Fig. 2 shows performance for growing n. It must be noted that the memory conservation plays a major role here. In case of slowly synchronizing automata the compact structures enable cache-friendly (therefore fast) computations for small n, while for larger n they are necessary for the algorithm to work, due to memory requirements. Performance of algorithm 3 supports this conclusion.

Table 1
Runtime (in seconds) for different implementations and different sizes of input Černý automata.

No	algorithm version	$\mathcal{C}_{12}$	$\mathcal{C}_{14}$	$\mathcal{C}_{18}$
0	TESTAS: SSW	< 1.0 s	18 s	Time-out
1	Set of int	0.34 s	3.3 s	60 s
2	hash_set of vector of int	0.07 s	0.34 s	9.7 s
3	hash_set of fast_bitset	0.01 s	0.07 s	2.0 s
4	Int array	0.0 s	0.01 s	0.2 s

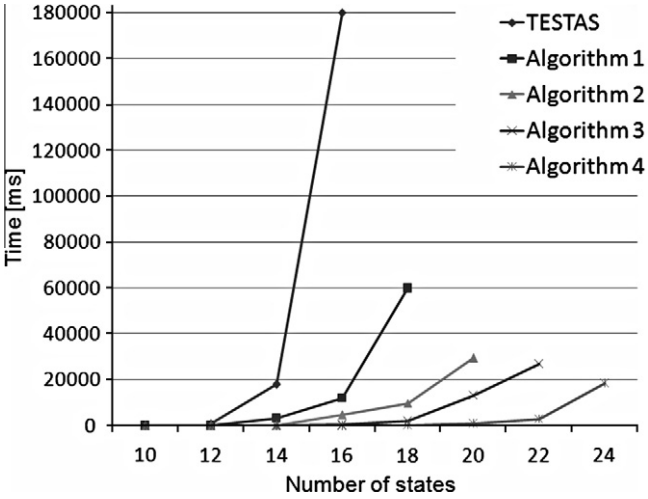

Fig. 2. Graphical representation of performance for $\mathcal{C}_n$.

Algorithm 3 should be as fast as standard implementations using power automaton. Unlike them, it can handle larger automata ($n > 30$) as long as the reachable power automaton is small.

Computations were performed with the use of a laptop manufactured in 2004 (Athlon XP-M 2800+(@2.13 GHz), 768 MB of DDR RAM (@133 MHz)).

6. New greedy algorithm

In this section we introduce a new greedy algorithm. It will be based on SYNCHROP, which itself is based on EPPSTEIN. Both algorithms are focused on choosing a pair of states, which should be synchronized in the next step and applying the proper word to the whole automaton. We can reformulate this task equivalently in terms of the pair automaton states (both EPPSTEIN and SYNCHROP use this structure) as choosing the state, which should be transformed to a singleton 0 state. From now on all procedures described below are regarded to the pair automaton $\mathcal{A}' = (Q', A, \delta')$ of the original input automaton $\mathcal{A} = (Q, A, \delta)$.

The choice of state depends on the set P of states that we are actually in, that is, the set $P = \delta'(Q', w)$, where w is the catenation of the words found in all previous steps. Such set will be called the *active states set*. The choice is based on the evaluation of the active states arrangement in the pair automaton. In SYNCHROP the evaluation is based on the following heuristics:

Heuresis 1. Let us define the distance $d(p)$ of state $p = \{s_1, s_2\}$ to the singleton state 0 as

$$d(p) = \min_{w \in \Sigma^*}(|w| : \delta(s_1, w) = \delta(s_2, w)). \tag{3}$$

Let w be a synchronizing word for some state q. The bigger is the difference between $d(p)$ and $d(\delta(p,w))$, the more profitable is the selection of w in the next algorithm step, because the distance of $\delta(p,w)$ to the singleton state is smaller.

The idea enclosed in the above heuristics is utilized in SYNCHROP in the following way: we compute the differences between states p and $\delta(p,w)$, where w is the shortest synchronizing word for the pair q, as

$$\Delta_q(p,w) = \begin{cases} d(\delta(p,w)) - d(p) & \text{if } p \neq q, \\ 0 & \text{if } p = q. \end{cases} \tag{4}$$

We compute $\Delta_q(p,w)$ for all active states except the singleton state. Let X be the set of all active states in the pair automaton. We define

$$\Phi_1(w) = \sum_{p\in X} \Delta_q(p,w). \tag{5}$$

Having $\Phi_1(w)$ for all the shortest words that synchronize pairs of states we choose the one with the smallest Φ_1 and apply it to the automaton. The modified version of SYNCHROP, SYNCHROPL, uses Φ_2 function – a modification of Φ_1 – which takes into account the length of the word:

$$\Phi_2(w) = \sum_{p\in X} \Delta_q(p,w) + |w| = \Phi_1(w) + |w|. \tag{6}$$

Thanks to this penalty component shorter words are preferable. This is a good solution in case where there are two or more candidate words with the same Φ_1 value.

Let us consider the complexity of SYNCHROP. The preprocessing part is the same as in EPPSTEIN and can be done in $O(n^3 + |A|n^2)$. The new part is the choice of s_1 and s_2, which are to be synchronized. This is done $O(n)$ times. To compute $\Phi_1(w)$ we have to process all active states in order to compute $\Delta((s_1,s_2),w)$ values, which are $O(n^2)$. The Φ_1 value has to be computed for all pairs of states of the input automaton ($O(n^2)$). Hence, the total complexity of the SYNCHROP is $O(n^3 + |A|n^2 + n(n^2 \cdot n^2)) = O(n^5 + |A|n^2)$.

6.1. FASTSYNCHRO – a better SYNCHROP

Comparing to NATARAJAN and EPPSTEIN, SYNCHROP and SYNCHROPL give good results, but have high complexity. In this section we present a modification of SYNCHROP with improved complexity and still preserving the quality. This algorithm will be called FASTSYNCHRO.

The first modification regarding to SYNCHROP is the way that synchronizing word is created. Instead of computing Φ for the words synchronizing pairs of states of $\mathcal{A}$, we will compute it for all letters from A. This modified function will be denoted by Φ_3. Then, we will choose the letter that minimizes Φ_3. This letter will be denoted by τ.

Let X be the set of active states in the pair automaton, A – the alphabet and $d(p)$ – the distance of p to the singleton state. We define

$$\Phi_3(l) = \sum_{p\in X} (d(\delta'(p,l)) - d(p)), \quad p \in X, l \in A,$$
$$\tau = \arg\min_{l\in A}(\Phi_3(l)).$$

Letter τ is applied to all active states and added at the end of the currently found word, increasing its length by 1. The drawback of this solution is that it does not guarantee us to finally find the synchronizing word – we do not know if the number of active states will decrease after some number of steps. Therefore, we will use Φ_3 only when it improves the arrangement of all active states in the pair automaton (that is, when $\Phi_3 < 0$). If $\Phi_3 \geqslant 0$, we will use Φ_2 function for finding the word that guarantees the synchronization, hence necessarily decreases the number of active states. However, we introduce one restriction. In SYNCHROP the greatest impact on the complexity has the computation of Φ for all pairs of states and all the shortest words that synchronize these pairs. This can be done in $O(n^4)$. We will reduce it to $O(n^3)$ by reducing the number of processed words from square to linear order of magnitude by choosing only n shortest words synchronizing the pairs (if there is less than n words, we choose all of them). Such a choice, inspired by EPPSTEIN, is simple in realization and gives better results in average than other choices of words that were checked by us.

The above modifications are given in Listing 8.

Algorithm 8 FASTSYNCHRO($\mathcal{A}$)

1: **Input**: automaton $\mathcal{A} = (Q, A, \delta)$
2: **Output**: synchronizing word w (if exists)
3: $w \leftarrow \varepsilon$
4: $\mathcal{A}' \leftarrow$pair automaton of $\mathcal{A}$
5: if $\mathcal{A}$ is not synchronizing **return** null
6: perform Eppstein preprocessing// see Section 4.4
7: $X \leftarrow Q$// X is the set of active states
8: count $\leftarrow 0$// a counter
9: **while** ($|X| > 1$)
10: $a \leftarrow \arg\min_{l\in A}\{\Phi_3(l)\}$
11: **if** ($\Phi_3(a) < 0$ AND count++ $< |Q|^2$)
12: $w \leftarrow w.a$; $X \leftarrow \delta(X,a)$
13: **else**
14: compute Φ_2 for $\min\{|Q|, \frac{|X|^2-|X|}{2}\}$ shortest words synchronizing the active states (denote Y for this set of words)
15: $v \leftarrow \text{argmin}_{y\in Y}\{|y|\}$
16: $w \leftarrow w.v$; $X \leftarrow \delta(X,v)$
17: **return** w

Let $n = |Q|$. The complexity of line 4. is $O(|A|n^2)$. In line 5. we check if an automaton is synchronizing. This requires the use of BFS on a pair automaton with reverse transitions. This takes $O(|A|n^2)$. Eppstein preprocessing in line 6. takes $O(|A|n^2 + n^3)$. Now consider the instructions in the **while** loop. The cost of line 10. is the cost of computing Φ_3 for all letters – $O(|A|n^2)$. Application of the letter or word to all active states (lines 12. and 16.) is $O(n)$, thanks to the Eppstein preprocessing. The cost of line 14. is $O(n^3)$, thanks to the restriction on the number of processed words.

Now it remains to compute the number of **while** loop calls. Each application of v to the set of active states reduces their number at least by 1, so lines 14. – 16. will be executed at most $n-1$ times. Alternatively, line 12. may be executed, but it does not necessarily reduce the number of active states. Therefore, to keep a tight rein on the total complexity, we set the restriction on the number of line 12. executions. In tests we have noticed that setting the limit to n^2 had no influence on the algorithm results.

Summarizing: the total cost of lines 4. – 6. is $O(|A|n^2 + n^3)$, the cost of the instructions inside the **while** loop is $O(n \cdot n^3) + O(n^2 \cdot |A|n^2)$. This gives us the following theorem.

Theorem 1. FASTSYNCHRO *works in* $O(|A|n^4)$ *time complexity.*

6.2. Experiments and comparison

In this section we present the results of the experiments on heuristic algorithms. We focus on the efficiency (the running time) and the quality (the length of the synchronizing word found). We also make some remarks on FASTSYNCHRO algorithm. We tested five heuristic algorithms: EPPSTEIN, CYCLE, SYNCHROP, SYNCHROPL and FASTSYNCHRO.

6.2.1. Efficiency

The efficiency has a great impact on the algorithms usability. In this subsection we present the efficiency comparison for EPPSTEIN, NATARAJAN, SYNCHROP and FASTSYNCHRO. Algorithms were tested for $n = \{10, 20, \ldots, 300\}$. For each n one hundred random automata were generated such that $\forall a \in A\ \forall p, q \in Q\ Pr(\delta(p,a) = q) = \frac{1}{n}$. If the generated automaton was not synchronizing, the procedure

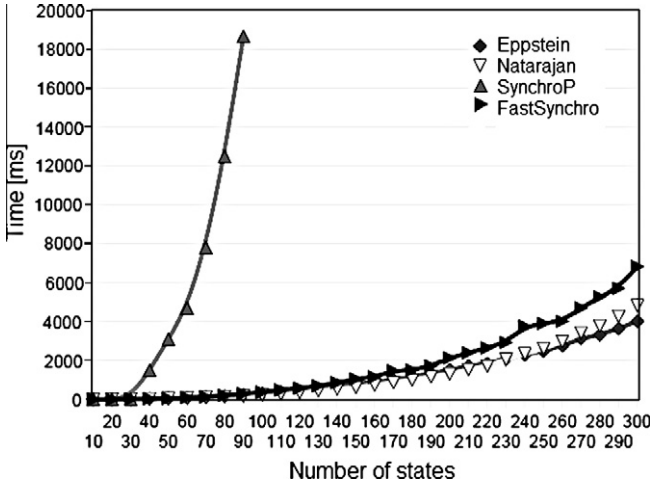

Fig. 3. Efficiency for automata with $|A| = 2$.

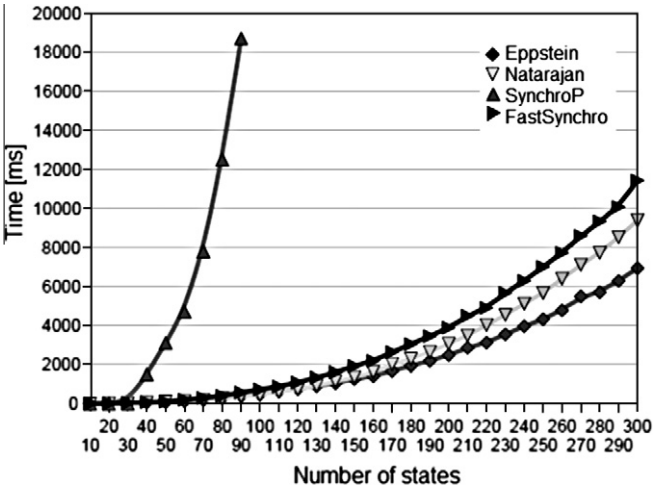

Fig. 4. Efficiency for automata with $|A| = 10$.

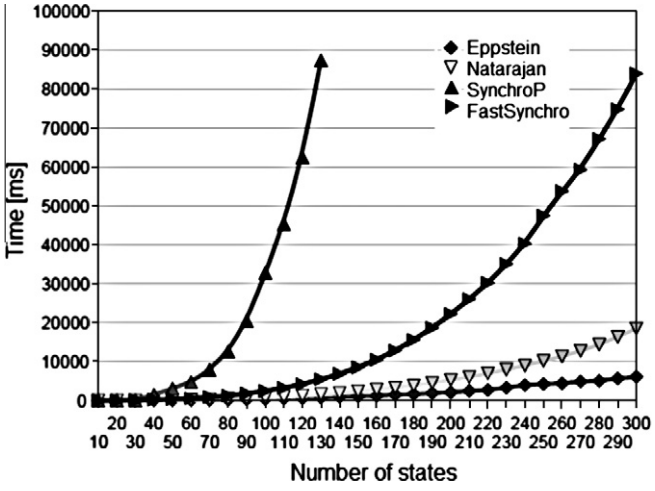

Fig. 5. Efficiency for Černý automata.

was repeated. Tests were performed for $|A| \in \{2,10\}$ (Figs. 3 and 4). We have also performed tests for Černý automata (Fig. 5).

It is clearly seen from Fig. 3 that SYNCHROP, due to its high complexity, is worse than other algorithms. The runtimes for other algorithms are comparable.

The results for $|A| = 10$ does not differ much from these with $|A| = 2$. Running times are of course higher, but still the differences between NATARAJAN, EPPSTEIN and FASTSYNCHRO are small.

Tests on Černý automata were performed to check how the algorithms work for automata with long synchronizing words. We can see noticeable decrease of efficiency in case of FASTSYNCHRO. EPPSTEIN and NATARAJAN work much faster in this case.

6.2.2. Quality

In order to compare the algorithms in the widest possible context, the tests were performed for three different quality measures. If the number of all automata for a given number of states and alphabet size was reasonable, the algorithms were tested on all such automata. If the number of such automata was too big, we reduced the tests to a subset of all possible random automata. All algorithms were run on the same set of automata.

The first quality measure is the mean difference between the length of the word found by the algorithm and the SSW length. Denote by $ALG(\mathcal{A})$ the word returned by algorithm ALG for automaton $\mathcal{A}$. Let X be the set of all automata that were given as the input to ALG. Formally, we can define the first measure as

$$M_1(X) = \frac{\sum_{\mathcal{A}\in X}(|ALG(\mathcal{A})| - |SSW(\mathcal{A})|)}{|X|}. \tag{7}$$

Table 2 shows that CYCLE and EPPSTEIN, despite their fast speed, does not give a good results in terms of M_1. SYNCHROP and SYNCHROPL, based on Heuristics 1, are much better. FASTSYNCHRO is comparable to them and sometimes outperforms them ($n = 5,6,10$).

The second quality measure, M_2 is the ratio of cases in which ALG found the SSW to all cases:

$$M_2(X) = \frac{\sum_{\mathcal{A}\in X}[|ALG(\mathcal{A})| = |SSW(\mathcal{A})|]}{|X|}, \tag{8}$$

where $[expr] = 1$ if $expr$ is true and 0 otherwise.

Notice how the quality decreases with increasing the alphabet size from 2 to 10. The ordering of the algorithms is the same as in case of M_1 measure.

To test the algorithms for automata with larger number of states, we need a measure which does not involve computing the

Table 2
Quality of algorithms in terms of M_1,M_2 and M_3. C, sc Ep, SP, SPL, FS correspond to CYCLE, EPPSTEIN, SYNCHROP, SYNCHROPL and FASTSYNCHRO algorithms.

$n,\|A\|$	C	EP	SP	SPL	FS	$\|X\|$
Measure M_1						
3,2	0.20	0.16	0	0	0	all automata
4,2	0.36	0.33	0.06	0.03	0.04	All automata
4,3	0.45	0.42	0.08	0.05	0.05	All automata
5,2	0.64	0.55	0.21	0.17	0.13	All automata
6,2	0.91	0.77	0.38	0.34	0.24	All automata
10,2	1.97	1.63	1.00	0.96	0.78	10^5 random automata
10,10	1.78	1.77	0.72	0.69	0.54	10^5 random automata
20,2	4.18	3.49	2.18	2.07	2.01	10^5 random automata
20,10	3.45	3.31	1.61	1.54	1.54	10^5 random automata
Measure M_2						
3,2	0.80	0.84	1	1	1	All automata
4,2	0.71	0.72	0.94	0.97	0.97	All automata
4,3	0.64	0.65	0.93	0.96	0.95	All automata
5,2	0.57	0.60	0.85	0.87	0.89	All automata
6,2	0.47	0.51	0.76	0.77	0.82	All automata
10,2	0.25	0.28	0.49	0.50	0.57	10^5 random automata
10,10	0.12	0.12	0.41	0.43	0.54	10^5 random automata
20,2	0.08	0.10	0.23	0.24	0.28	10^5 random automata
20,10	0.02	0.02	0.13	0.14	0.16	10^5 random automata
Measure M_3						
50,2	26.21	24.44	21.75	21.53	21.93	10^4 random automata
50,10	16.32	15.49	12.84	12.71	13.00	10^4 random automata
100,2	40.75	37.53	33.16	32.84	33.95	10^3 random automata
100,10	25.30	23.41	19.84	19.61	20.78	10^3 random automata

Table 3
Behavior of FASTSYNCHRO for random automata.

$n,\|A\|$	η_1	η_2	λ	λ^*	$\lambda-\eta_1$	$\frac{\lambda-\eta_1}{\eta_2}$	$\|X\|$
10,2	6.52	0.44	7.52	6.92	0.99	2.28	10 000
20,2	10.11	0.77	12.26	10.13	2.15	2.80	10 000
50,2	16.12	1.63	22.07	16.77	5.95	3.65	10 000
100,2	21.69	2.83	34.07	24.55	12.38	4.38	1 000
200,2	27.38	4.56	51.81	35.94	24.42	5.35	1 000
10,10	4.01	0.02	4.04	n/a	0.02	1.00	10 000
20,10	6.68	0.21	6.91	n/a	0.22	1.07	10 000
50,10	12.00	0.76	13.02	n/a	1.01	1.34	10 000
100,10	17.60	1.92	20.88	n/a	3.28	1.71	1 000
200,10	24.32	4.16	32.72	n/a	8.40	2.02	1 000

Table 4
Behavior of FASTSYNCHRO for Černý automata.

n	η_1	η_2	λ	$\lambda-\eta_1$	$\frac{\lambda-\eta_1}{\eta_2}$
10	24	3	81	57	19
20	67	4	361	294	73
50	211	5	2401	2190	438
100	520	6	9801	9281	1546
200	1238	7	39601	38363	5480

SSW length. Therefore, as M_3 we took the mean length of the synchronizing words found by a given ALG. The use of this measure is meaningful only in relative comparison of two or more algorithms.

$$M_3(X) = \frac{\sum_{\mathcal{A}\in X}|ALG(\mathcal{A})|}{|X|}. \quad (9)$$

FASTSYNCHRO, in terms of M_3, gives a little worse results than SYNCHROP and SYNCHROPL, however these results are still much better than those of EPPSTEIN and CYCLE.

6.3. Analysis of FASTSYNCHRO behavior

In this section we investigate the behavior of FASTSYNCHRO. The analysis will allow us to explain the decrease of FASTSYNCHRO efficiency shown in Fig. 5. We will check what is the impact of different algorithm's parts on the process of building the synchronizing word. Recall that in FASTSYNCHRO a synchronizing word can be obtained in two ways: first one is choosing the letter $a \in A$ and apply it to the set of active states. The other is to use Φ_2 to find the pair of states and transform the set of active states by the word synchronizing this pair. We will refer to these two ways as to the first and the second part of the algorithm.

We have performed an experiment for random automata. By η_1 (resp. η_2) we denote the number of executions of the first (resp. second) part of the algorithm. By λ we denote the mean length of the synchronizing word found by FASTSYNCHRO and by λ^* the estimated value of the SSW length for random automata over binary alphabet (Skvortsov & Tipikin, 2011).

Notice that each execution of the first part of the algorithm corresponds to the generation of exactly one letter added to the synchronizing word that is constructed. The value $\lambda-\eta_1$ expresses the number of letters added in result of the second part execution. From Table 3 we can see that with the increase of the number of states the fraction of the second part execution also grows.

The ratio $\frac{\lambda-\eta_1}{\eta_2}$ is the mean length of the word added during the second part execution. This value remains relatively small and grows slightly with the increase of the number of states.

When $|A|$ is increased to 10, we can see that the influence of the second part decreases. Despite that its frequency is almost the same as in case $|A| = 2$, the mean length of the word added by each execution is 2–3 times shorter.

The same experiment was performed for Černý automata (Table 4).

These experiments explain why the runtime depends so much on the length of the synchronizing word found by the algorithm: the number of executions of the algorithm's first part increases significantly. Also, the words found by the execution of the algorithm's second part are not short anymore. Fortunately, the automata with long SSWs are very rare, so the case of Černý automata is exceptional.

7. Conclusions

We presented some efficient data structures for exact (exponential) synchronizing algorithm. Their application into the well-known algorithm that uses a power-set automaton makes the algorithm more effective than existing implementations. We also presented a new, greedy synchronizing algorithm and we compared it with some previously known greedy algorithms. Experiments show that our FASTSYNCHRO algorithm in general works better (that is, finds shorter synchronizing words) and usually works in a comparable time or faster than other methods. For larger automata FASTSYNCHRO works twice as long as EPPSTEIN, but it finds much shorter synchronizing words.

When one wants to find a synchronizing word, two factors have to be considered: the quality (the length of the synchronizing word) and the time. If time is a key issue, the optimal choice would be the EPPSTEIN algorithm. But if the quality is much more important (and this is usually the case in industrial testing of electronic circuits, when one has to apply the same synchronizing word to thousands or millions copies of circuits), the best choice is to use our new FASTSYNCHRO algorithm.

References

Ananichev, D. S., & Volkov, M. V. (2003). Synchronizing monotonic automata. *Lecture Notes in Computer Science, 2710*, 111–121.

Björn, K. (2005). *Beyond the C++ standard library: An introduction to boost*. Addison-Wesley.

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., & Pretschner, A. (2005). Model-based testing of reactive systems model-based testing of reactive systems. Advanced lectures. *Lecture Notes in Computer Science*, 3072.

Černý, J. (1964). Poznámka k. homogénnym experimentom s konecnymi automatmi. *Matematicko-fyzikálny Časopis Slovenskej Akadémie Vied, 14*, 208–215.

Černý, J., Pirická, A., & Rosenauerová, B. (1971). On directable automata. *Kybernetika, 7*(4), 289–298.

Deshmukh, R. G., & Hawat, G. N. (1994). An algorithm to determine shortest length distinguishing, homing, and synchronizing sequences for sequential machines. In *Proc. Southcon 94 conference* (pp. 496–501).

Eppstein, D. (1990). Reset sequences for monotonic automata. *SIAM Journal of Computing, 19*(3), 500–510.

Fukada, A., Nakata, A., Kitamichi, J., Higashino, T. & Cavalli, A. R. (2001). A conformance testing method for communication protocols modeled as concurrent dfsms. In *ICOIN* (pp. 155–162).

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

Hyunwoo, C., Somenzi, F., & Pixley, C. (1993). Multiple observation time single reference test generation using synchronizing sequences. In *Proc. IEEE European conf. on design automaton* (pp. 494–498).

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

Klyachko, A. A., Rystsov, I. K., & Spivak, M. A. (1987). An extremal combinatorial problem associated with the bound of the length of a synchronizing word in an automaton. *Cybernetics and Systems Analysis, 23*(2). translated from Kibernetika, No 2, 1987, pp. 16–20, 25.

Lee, D., & Yannakakis, M. (1996). Principles and methods of testing finite state machines – a survey. In *Proceedings of the IEEE* (Vol. 84, pp. 1090–1123).

Natarajan, B. K. (1986). An algorithmic approach to the automated design of part orienters. In *Proc. IEEE symposium on foundations of computer science* (pp. 132–142).

Olschewski, J., & Ummels, M. (2010). The complexity of finding reset words in finite automata. *Lecture Notes in Computer Science, 6281*, 568–579.

Pin, J.-E. (1983). On two combinatorial problems arising from automata theory. *Annals of Discrete Mathematics, 17*, 535–548.

Pixley, C., Jeong, S.-W., & Hachtel, G. D. (1994). Exact calculation of synchronizing sequences based on binary decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13*(8), 1024–1034.

Pomeranz, I., & Reddy, S. M. (1998). On synchronizing sequences and test sequence partitioning. In *Proc. 16th IEEE VLSI test symposium* (pp.158–167).

Ponce, A.M., Csopaki, G., & Tarnay, K. (1994). Formal specification of conformance testing documents for communication protocols. In *5th IEEE international symposium on personal, indoor and mobile radio communications* (Vol. 4, pp. 1167–1172).

Roman, A. (2009). Synchronizing finite automata with short reset words. *Applied Mathematics and Computation, 209*(1), 125–136.

Skvortsov, E., & Tipikin, E. (2011). Experimental study of the shortest reset word of random automata. *Lecture Notes in Computer Science, 6807*, 290–298.

Trahtman, A. N. (2006). An efficient algorithm finds noticeable trends and examples concerning the cerny conjecture. *Lecture Notes in Computer Science, 4162*, 789–800.

Trahtman, A. N. (2009). The road coloring problem. *Israel Journal of Mathematics, 1*(172), 51–60.

Volkov, M. V. (2008). Synchronizing automata and the cerny conjecture. *Lecture Notes in Computer Science, 5196*, 11–27.

Zhao, Y., Liu, Y., Guo, X., & Zhang, C. (2010). Conformance testing for is-is protocol based on e-lotos. In *IEEE int. conf. on information theory and information security* (pp. 54–57).