A New Heuristic Synchronizing Algorithm

Jakub Kowalski*, Marek Szykula*

Department of Mathematics and Computer Science, University of Wroclaw
{kot,msz}@i2i.uni.wroc.pl

Abstract. We present a new heuristic algorithm finding reset words. The algorithm called CutOff-IBFS is based on a simple idea of inverse breadth-first-search in the power automaton. We perform an experimental investigation of effectiveness compared to other algorithms existing in literature, which yields that our method generally finds a shorter word in an average case and works well in practice.

Keywords: Synchronizing automata, synchronizing algorithm, reset word

1 Introduction

Synchronizing automata are important in various fields, such as model-based testing [5], robotics (for designing so-called part orienters) [2], bioinformatics (the reset problem) [3], network theory [7], theory of codes [8] etc. In many applications it is important to find a reset word as short as possible. Unfortunately the problem of finding a shortest reset word is shown to be FP$^\text{NP}[\log_2]$-hard, and the related decision problem is both NP- and coNP-hard [14] (cf. also [4] and [12,13] for approximation hardness and special classes).

Nevertheless there exist many exponential algorithms to deal with this problem [11,15,18,19,20] and a lot of polynomial heuristics finding relatively the shortest reset words [11,19,20]. Recently we developed an exact algorithm based on a bidirectional-breadth-first search, which is currently the fastest algorithm for the problem [10]. Our new heuristic algorithm was also used as a part of our exact algorithm.

The algorithm, called CutOff-IBFS, is a heuristic polynomial synchronizing algorithm finding a synchronizing word within the given length. We analyzed some properties of the algorithm and performed an experimental investigation of the algorithm and compared with some others existing in literature. Our method, although simple, seems to be more efficient in practice and generally finds shorter reset words in less time. For simplicity, we describe only the version which find the length of the found word, and later consider a suitable modification to find the word itself.

2 Description of the Algorithm

Before we run the main part of the algorithm we must run some other algorithm to obtain a reset word and an upper bound on the length of the shortest reset words. This is necessary as we do not have a guarantee that it finds any word in all cases. To have this property it can be easily combined with some other algorithm. We use well-known Eppstein algorithm [6] as a preceding algorithm. Then we apply the length of the found word into CutOff-IBFS as maxlen. It is preferred to use a

* Supported in part by Polish MNiSZW grant IP2012 052272
fast preceding algorithm with not greater complexity than CutOff-IBFS, to not increase the overall complexity.

The formal description is given in Algorithm 1. We start from the list $L$, which contains all singletons of the states. Then in each step we create the set of preimages of the sets from $L$ by each letter. The sets are stored in the trie $T_{ic}$, which allows us to exclude all duplicated sets. Next we create the resulted list $L'$ by taking only the maxsize largest sets from $T_{ic}$. We repeat the steps until we obtain the complete set or we have done maxlen. In the second case we use the word found by the preceding algorithm as a result.

Algorithm 1 CutOff-IBFS

Require: $A = (Q, \Sigma, \delta)$ – an automaton with $n = |Q|$ states and $k = |\Sigma|$ input letters.

Require: maxlen – maximum length of words to be checked.

Require: maxsize – maximum size of the lists.

1: procedure CutOff-IBFS($A$, maxlen, maxsize)
2: $L \leftarrow \text{EmptyList}$
3: for all $q \in Q$ do
4:  $L$.insert($\{q\}$)
5: end for
6: for $l \leftarrow 1$ to maxlen do
7:  $T_{ic} \leftarrow \text{EmptyTrie}$ \Comment{A trie to store next sets}
8:  for all $S \in L$ do
9:      for all $a \in \Sigma'$ do \Comment{Compute the image of $S$ by $a$}
10:         $S' = \delta(S, a)$
11:         if $|S'| = n$ then \Comment{The goal test}
12:            return $l$
13:         end if
14:      $T_{ic}$.insert($S'$)
15:  end for
16: end for
17: $L' \leftarrow \text{EmptyList}$
18: $L'$.insert($T_{ic}$)
19: Sort $L'$ by descending cardinalities.
20: Trim $L'$ to maxsize length, by truncating from the end.
21: $L \leftarrow L'$
22: end for
23: return “Not found a synchronizing word of length $\leq$ maxlen”
24: end procedure

2.1 Analysis

The correctness of the algorithm comes from the fact that we can assign a word to each computed set, so that an image of a set by its assigned word is a singleton. The word assigned to the final set $S'$ of size $n$ would be a synchronizing one.

Complexity of the algorithm depends of the given maxsize parameter. The larger maxsize is the slower the algorithm works and uses more memory, but it finds shorter words. Complexity of
the algorithm depends linearly on \( c \) so it is easy to set a desirable trade-off. Also it works fast when it finds a short word, and this is the case for random automata. A trie is used to skip sets already stored in it, keeping insertion time in \( O(n) \).

**Theorem 1.** CutOff-IBFS works in \( O(lckn + kn^2) \), where \( l \) is the length of the found reset word and \( c = \maxsize \geq 1 \). The space complexity is \( O(ckn) \).

**Proof.** Initialization in lines 2–5 takes \( O(n^2) \) time (\( n \) sets). We assume now the worst case, that the word is not found and the for loop in line 6 runs \( l \) times. During all the executions except the first we have at most \( c \) sets in \( L \). Computing preimages of the sets (line 10) by each of the letter can be done in \( O(ckn) \) time (\( O(n) \) for each preimage). Insertion the preimages into a trie (line 14) is done in \( O(ckn) \) time (\( O(n) \) for an insertion). Sorting \( L \) (line 19) can be done in \( O(ckn) \) time by counting sort. During the first execution of the for loop there are \( n \) sets at the beginning, so it takes \( O(kn^2) \) time. The total time complexity is \( O(lckn + kn^2) \). Space is required to store the list \( L \) and the trie \( T_c \) during all steps, which yields \( O(ckn) \).

Together with preceding Eppstein algorithm (see [6], it works in time \( O(kn^2 + n^3) \) and \( O(kn^2) \) space), and taking \( l \in O(n^3) \), the algorithm has \( O(ckn^4) \) time complexity. The total space complexity is \( O(ckn + kn^2) \).

Inverse breadth-first-search is very effective for some of the most extremal automata (with the longest reset lengths). Similar as our exact algorithm [10] works in polynomial time for them, our heuristic algorithm always find the shortest reset word in these cases.

**Theorem 2.** When \( \maxsize \geq n \), Algorithm [1] always finds the shortest reset word for the Černý automaton \( \mathcal{C}_n \) and the slowly synchronizing series introduced in [11] \( \mathcal{D}_n, \mathcal{W}_n, \mathcal{F}_n, \mathcal{E}_n, \mathcal{D}_n', \mathcal{R}_n, \mathcal{J}_n, \mathcal{H}_n \).

**Proof.** Consider the automaton \( \mathcal{C}_n \). We will show that a set corresponding with the shortest reset word is kept in the list during all executions of the main loop in line 6, that is after \( i \) steps there will be a set \( S \) which is a preimage of some singleton by the word consisted of the \( i \) last letters of the shortest reset word. This is true at the beginning since we have all the singletons in the list.

After the first step (the first execution of the main loop in line 7) there will be a set of size 2 in the list and the singletons, since it is kept because of the sorting in line 19. After each of the \( (n-1) \) next steps there is added a new set of size 2 and all of them are kept since there are only \( n \) such sets after the \( n \)-th step. In the \( (n+1) \)-th step a set of size 3 is created and it is the largest set in the list. We can continue in this way and after each of part of \( (n-1) \) steps there will be a step introducing a new set of size greater by 1, while within a part each step introduces exactly one set of the currently largest size. There are \( (n-1) \) steps introducing a larger set and \( (n-2) \) parts consisted of \( (n-2) \) steps. So in the \( ((n-1)^2) \)-th step there will be introduced the final set of size \( n \).

In a similar way one may follow execution of the algorithm for the other automata series to see that a set corresponding with the shortest reset word is kept in the list, and so the algorithm finds the shortest reset word.

### 2.2 Finding a Reset Word

The algorithm can also return the found word, not only its length. It however increases space complexity, but the time complexity can be kept. We cannot store complete words together with the sets since they may be long \( (l) \), so creating a new set would take \( O(n + l) \) time due to copying.
issue. Instead we can store, together with a set, the applied letter and a pointer to the information stored for the preceding set. This additional information costs a constant space for a set and can be computed in a constant time for a preimage set. However we need to preserve this information for each constructed set, so it will take additional $O(cl + n)$ space. This would yield total space complexity $O(cl + kn^2)$ space complexity, while keeping time complexity in $O(lckn + kn^2)$.

### 2.3 Technical Improvements

We discuss here some technical improvements of the algorithm. They can reduce running time and space but within a constant factor.

First of all, we can skip sorting from the main loop. Instead we can maintain $n$ tries for each possible cardinality of a set. Then we can construct a list by taking sets from these tries from the largest to the smallest.

Another simple improvement comes from the fact, that we can start only from the singletons of states from the sink component, as well as only from the states having in-degree at least 2 on some letter.

One may also permute the automaton before running the algorithm. Since the sets during computation are usually small, it could be better to have states occurring more frequently to be at the top of tries. It would lead to have smaller heights of the tries and so to faster execution. A simple heuristic method to do that is sorting the states decreasing by their in-degrees.

### 3 Empirical Behavior

![Graph](image)

Fig. 1. The mean found length by the algorithms for random automata.
We have compared some known synchronizing algorithms in terms of the quality (found word lengths) and the execution time. The experiment was performed on 10,000 uniformly random labeled automata for \( n = 100, 200, \ldots, 1000 \) (each algorithm worked on the same automata set). Figure 1 shows the mean length found by the algorithms which is generally the quality measure. Figure 2 shows the average execution time used by the algorithms.

We presented two versions of CutOff-IBFS, with \( c = \log n \) and \( c = n \). See [17,11] for the other algorithms. We can see that CutOff-IBFS found shorter reset words than the other algorithms and the \( \log n \) version is slower only than two fastest Eppstein and Cycle algorithms.

Since there may be different versions of these algorithms or they may be run with different parameters, which can affect the results of the experiment, we describe here some implementation details. For CutOff-IBFS we included time used by Eppstein algorithm. We used the greedy versions of Eppstein algorithm [6] and Cycle algorithm [17,11], which always selects a pair of states with the shortest synchronizing word. For the tree of the pair automaton we used \( n \) as the constant used for marking nodes (this is \( y \) constant in Algorithm 2 from [6]).

References

1. D. Ananichev, V. Gusev, and M. Volkov. Primitive digraphs with large exponents and slowly synchronizing automata. *Zapiski Nauchnyh Seminarov POMI [Kombinatorika i Teorija Grafov. IV]*, 402:9–39, 2012. In Russian.
2. D. Ananichev and M. Volkov. Synchronizing monotonic automata. In *Developments in Language Theory*, volume 2710 of *LNCS*, pages 111–121. 2003.
3. Y. Benenson, R. Adar, T. Paz-Elizur, Z. Livneh, and E. Shapiro. DNA molecule provides a computing machine with both data and fuel. *Proceedings of the National Academy of Sciences*, 100(5):2191–2196, 2003.
4. M. Berlinkov. Approximating the minimum length of synchronizing words is hard. In Computer Science – Theory and Applications, volume 6072 of LNCS, pages 37–47. 2010.
5. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science). Springer-Verlag New York, Inc., 2005.
6. D. Eppstein. Reset sequences for monotonic automata. SIAM Journal on Computing, 19:500–510, 1990.
7. M. Gerbush and B. Heeringa. Approximating minimum reset sequences. In Implementation and Application of Automata, volume 6482 of LNCS, pages 154–162. 2011.
8. Jürgensen, H. Synchronization. Information and Computation, 206(9-10):1033–1044, 2008.
9. J. Kari. Synchronization and stability of finite automata. Journal of Universal Computer Science, 8(2):270–277, 2002.
10. A. Kisielewicz, Kowalski J., and Szykuła M. A Fast Algorithm Finding the Shortest Reset Words. In Computing and Combinatorics, volume 7936 of LNCS, pages 182–196, 2013.
11. R. Kudłacik, A. Roman, and H. Wagner. Effective synchronizing algorithms. Expert Systems with Applications, 39(14):11746–11757, 2012.
12. P. Martyugin. Complexity of problems concerning reset words for some partial cases of automata. Acta Cybernetica, 19:517–536, 2009.
13. P. Martyugin. Complexity of problems concerning reset words for cyclic and Eulerian automata. In Implementation and Application of Automata, volume 6807 of LNCS, pages 238–249. 2011.
14. J. Olschewski and M. Ummels. The complexity of finding reset words in finite automata. In Mathematical Foundations of Computer Science 2010, volume 6281 of LNCS, pages 568–579. 2010.
15. J.-K. Rho and Somenzi F. Minimum length synchronizing sequences of finite state machine. In Proceedings of the 30th ACM/IEEE Design Automation Conference, DAC ’93, pages 463–466, 1993.
16. A. Roman. Genetic algorithm for synchronization. In Language and Automata Theory and Applications, volume 5457 of LNCS, pages 684–695. 2009.
17. A. Roman. Synchronizing finite automata with short reset words. Applied Mathematics and Computation, 209(1):125–136, 2009.
18. S. Sandberg. Homing and synchronizing sequences. In Model-Based Testing of Reactive Systems, volume 3472 of LNCS, pages 5–33. 2005.
19. E. Skvortsov and E. Tipikin. Experimental study of the shortest reset word of random automata. In Implementation and Application of Automata, volume 6807 of LNCS, pages 290–298. 2011.
20. A. N. Trahtman. An efficient algorithm finds noticeable trends and examples concerning the Černý conjecture. In Mathematical Foundations of Computer Science, volume 4162 of LNCS, pages 789–800. 2006.