An improved algorithm for finding the shortest synchronizing words

(Ulepszony algorytm znajdujący
najkrótsze słowa synchronizujące)

Adam Zyzik

Praca licencjacka

Promotor: dr Marek Szykuła

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

11 lutego 2022

Abstract

A synchronizing word of a deterministic finite automaton is a word whose action maps every state to a single one. Finding a short or a shortest synchronizing word is a central computational problem in the theory of synchronizing automata. It is also applied in areas such as model-based testing and the theory of codes. Because the problem of finding a shortest synchronizing word is computationally hard, among *exact* algorithms only exponential ones are known. There also exist polynomial heuristics designed in the hope of finding relatively short reset words in practice. In this thesis, we redesign and improve the best known exact algorithm. This is obtained with a combination of new algorithmic and technical enhancements. Next, we develop a general computational package devoted to the problem, where an efficient and practical implementation of our algorithm is included, together with several well-known heuristics. The package supports just-in-time compilation, multithreaded and GPU computing, and configurable computation plans. Our experiments show that the new algorithm is multiple times faster than the previously fastest one and its advantage quickly grows with the hardness of the problem instance. Given a modest time limit, we could compute the lengths of the shortest reset words for random binary automata up to 500 states, significantly beating the previous record.

Słowem synchronizującym deterministyczny automat skończony nazywamy takie słowo, którego działanie przenosi wszystkie stany do jednego ustalonego. Znajdowanie krótkiego lub najkrótszego słowa synchronizującego jest centralnym problemem obliczeniowym w teorii automatów synchronizowalnych. Ma także zastosowania w innych obszarach, takich jak testowanie bazujące na modelach i teoria kodów. Ponieważ problem znalezienia najkrótszego słowa synchronizującego jest trudny obliczeniowo, spośród algorytmów *dokładnych* znane są tylko wykładnicze. Istnieją również wielomianowe heurystki projektowane w nadziei na znajdowanie relatywnie krótkich słów w praktyce. W tej pracy ulepszamy najlepszy znany dotąd algorytm dokładny. Wykorzystujemy w tym celu kombinację nowych algorytmicznych i technicznych usprawnień. Następnie tworzymy ogólny pakiet obliczeniowy poświęcony temu problemowi, zawierający efektywną i praktyczną implementację naszego algorytmu wraz kilkoma znanymi algorytmami heurystycznymi. Pakiet wspiera kompilację just-in-time, obliczenia wielowątkowe i na GPU, oraz konfigurowalne plany obliczeniowe. Nasze eksperymenty pokazują, że nowy algorytm jest wielokrotnie szybszy od najszybszego dotychczas znanego algorytmu, a jego przewaga rośnie wraz z trudnością instancji problemu. Zakładając dosyć mały limit czasowy, byliśmy w stanie obliczyć długości najkrótszych słów synchronizujących dla losowych automatów binarnych mających do 500 stanów, znacznie pokonując poprzedni rekord.

Contents

Chapter 1

Introduction

A *deterministic finite complete semi-automaton* (called simply an *automaton*) is a 3-tuple (Q, Σ, δ), where Q is a finite set of *states*, Σ is an *input alphabet*, and $\delta\colon Q \times \Sigma \to Q$ is the *transition function.* The transition function is naturally extended to a function $Q \times \Sigma^* \to Q$. By n we denote the number of states in Q and by k we denote the size of the input alphabet $|\Sigma|$.

A word is *reset* (or *synchronizing*) if $|\delta(Q, w)| = 1$; in other words, for every two states $p, q \in Q$ we have $\delta(q, w) = \delta(p, q)$. An automaton that admits a reset word is called *synchronizing.* Then, its *reset threshold* is the length of the shortest reset words.

The classical synchronization problem is, for a given automaton, to find a reset word. Preferably, this word should be as short as possible. Therefore, the main property of a synchronizing automaton is its reset threshold.

Synchronizing automata and the synchronization problem are known for both their interesting theoretical properties and practical applications.

1.1 Theoretical developments

On the theoretical side, there is a famous long-standing open problem from 1969 [36] called the Černý conjecture (see an old [36] and a recent survey [14]). It claims that the reset threshold is at most $(n-1)^2$. If true, the bound would be tight [6].

Until 2017, the best known upper bound on the reset threshold was $(n^3 - n)/6 - 1 \sim 0.1666\ldots n^3 + \mathcal{O}(n^2)$ $(n \geq 4)$ [21] by the well-known Frankl-Pin's bound. The current best known upper bound is $\sim 0.1654n^3 + o(n^3)$ by Shitov [30], which was obtained by refining the previous improvement $\sim 0.1664n^3 + \mathcal{O}(n^2)$ by Szykuła [32]. Apart from that, better bounds were obtained for many special subclasses of automata and several new results around the topic appear every year. Recently, a special journal issue was dedicated to the problem [37] for the occasion of the

50th anniversary of the problem. Synchronizing automata are also applied in other theoretical areas, e.g., matrix theory [10], theory of codes [5], Markov processes [35].

Reset thresholds were also studied for the average case. Berlinkov has shown that a random binary automaton is synchronizing with high probability [3]. Moreover, Nicaud has shown that an automaton with high probability has a reset threshold in $\mathcal{O}(n \log^3 n)$ [19]. Based on that, the upper bound $\mathcal{O}(n^{3/2+o(1)})$ on the expected reset threshold of a random binary automaton has been obtained [2]. These studies were accompanied by experiments, and the best estimation obtained so far is $2.5\sqrt{n-5}$ [15, 16].

1.2 Synchronization in applications

Apart from being a theoretical problem, the synchronization problem can be applied in practical areas, e.g., testing of reactive systems [24, 27], networks [13], robotics [1], and codes [11].

In more detail, automata are frequently used to model the behavior of systems, devices, circuits, etc. The idea of synchronization is natural: we aim to restore control over a device whose current state is not known. For instance, for digital circuits, where we need to test the conformance of the system according to its model, each test is an input word and before we run the next one, we need to restart the device. In another setting, the observer wants to eventually learn the current state of the automaton by observing the input; once a reset word appears, the state is revealed. See a survey [27] explaining synchronizing sequences and their generalization to automata with output: *homing sequences*, which are used to determine the (hidden) current state of an automaton.

Another particular application comes from the theory of codes, where finite automata act as *decoders* of a compressed input. Synchronizing words can make a code resistant to errors, since if an error occurs, after reading such a word decoding is certainly restored to the correct path. See a book for the role of synchronization in the theory of codes [5] and recent works [4] explaining synchronization applied to prefix codes.

1.3 Algorithms finding reset words

The problem of finding a short or a shortest reset word is central for both theory and practice. Obviously, it is better to have reset words as short as possible.

Unfortunately, determining the reset threshold is computationally hard. The decision problem is NP-complete [7] and remains hard even for such a restrictive class as binary Eulerian automata [38]. The functional problems of computing the length and a shortest reset word are respectively $\mathrm{FP}^{\mathrm{NP}[\log]}$-complete and $\mathrm{FP}^{\mathrm{NP}}$-complete [20]. Moreover, approximating the reset threshold is hard even for approximation factors in $\mathcal{O}(n^{1-\varepsilon})$, for every $\varepsilon > 0$ [8]. This inapproximatibility also holds for subclasses related to recognizing prefix codes [26]. On the other hand, there exists a simple general $\mathcal{O}(n)$-approximating algorithm [9].

Therefore, either exponential exact algorithms are used or polynomial heuristics that hopefully find a relatively short reset word in typical cases.

1.3.1 Exact algorithms

Exact algorithms can be used for automata that are not too large. They also play an important role in testing heuristics, providing the baseline for comparison (e.g., [25]).

The standard algorithm, e.g., [18, 27, 34], for computing a shortest reset word is finding a path in the power automaton (whose set of states is $2^Q \setminus \emptyset$) from Q to a singleton. It requires $\mathcal{O}(2^n)$ space, which is acceptable only up to very small $n \sim 30$.

Alternative approaches include utilizing SAT solvers [31] by suitable reductions. This was also recently tried for partial deterministic finite automata [29]. In the reported results, solutions based on SAT solvers reach random binary automata with about 100 states.

Nevertheless, the fastest known algorithm was based on a bidirectional search of the power automaton, equipped with several enhancements [15, 16]. This algorithm was able to deal with binary random automata up to 350 states. Despite several attempts, no faster solutions were developed and the algorithm was not improved until now.

1.3.2 Heuristic algorithms

The most classic algorithm is Eppstein's one [7]. It works in $\mathcal{O}(n^3+n^2k)$ time, where n is the number of states and k is the size of the alphabet, and finds a reset word of length at most $(n^6-n)/3$ (due to the Frankl-Pin's bound). Several improvements were proposed, e.g., Cycle, SynchroP, and SynchroPL algorithms [18, 34], which do not improve guarantees but behave better in experimental settings.

Recent works also involve attempts to speed-up heuristics by adapting to parallel and GPU computation [28, 33].

A remarkable heuristic is the *beam* algorithm based on inverse BFS ([25, *CutOff-IBFS*]), which significantly beats other algorithms based on the forward search (yet, curiously, it does not provide any worst-case guarantees and it may not find any reset word at all).

Alternative approaches involve, e.g., hierarchical classifier [23], genetic algorithms [17], and machine learning approaches [22].

1.4 Contribution

We reinvestigate the best known exact and heuristic algorithms and provide their effective implementation.

We take the almost 10-years old exact algorithm based on bidirectional search [15, 16] and significantly improve it. We develop a series of algorithmic enhancements involving better data structures, decision mechanisms, and reduction procedures. Additionally, the remodeled algorithm is adapted for effective usage of multithreading and GPU utilization, which was difficult in the original version due to large shared data structures.

In the implementational part, we provide an open computational package containing the new exact algorithm as well as several polynomial heuristics. We apply a series of technical optimizations in order to maximize efficiency. The package is written in C++17, uses just-in-time (JIT) compilation, and can be easily extended with new algorithms. We apply original solutions as configurable JIT computation plans.

Altogether, we obtain a significant speed-up and decrease the memory requirements. In the experimental chapter, we compute the reset thresholds of binary random automata up to 500 states.

Chapter 2

The new exact algorithm

2.1 Overview

The algorithm is based on the former best exact algorithm [15, 16]. While the new version differs in the choice of data structures and subprocedures, at high level it is similar and uses two main phases – bidirectional breadth-first search and then inverse depth-first search in the power automaton. We start with a few auxiliary definitions.

Definition 2.1. 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 $\delta^{-1}(S, w) = \{q \in Q \mid \delta(q, w) \in S\}$.

Definition 2.2. The *power automaton* of $\mathscr{A} = (Q, \Sigma, \delta)$ is the automaton $\mathcal{P}(\mathscr{A}) = (\mathcal{P}(Q), \Sigma, \delta')$, where $\delta'(X, a) = \delta(X, a)$ (the image of X under the action of a), for all $X \subseteq Q$, $a \in \Sigma$.

Figure 2.1 shows an example of the power automaton of the Černý automaton [6] with 4 states.

The input to the algorithm is an automaton $\mathscr{A} = (Q, \Sigma, \delta)$ with n states and k input letters. In the first step, we check if $\mathscr{A}$ is synchronizing and get an upper bound on the reset threshold by using the polynomial-time Eppstein algorithm [7]. We then try to strengthen this bound with various polynomial-time heuristics, as it may help the main procedure make better decisions. Next, we proceed to the main part.

The key idea is to simultaneously run a breadth-first search (BFS) starting from the set Q and computing images, together with an inverse breadth-first search (IBFS) starting from all of the singletons and computing preimages. While both algorithms on their own require computation of at most k^r or at most nk^r sets, respectively, where r is the reset threshold, combining them lets us compute no more than $nk^{r/2}$ sets, provided that we can somehow test if the searches have met. To do this, we need

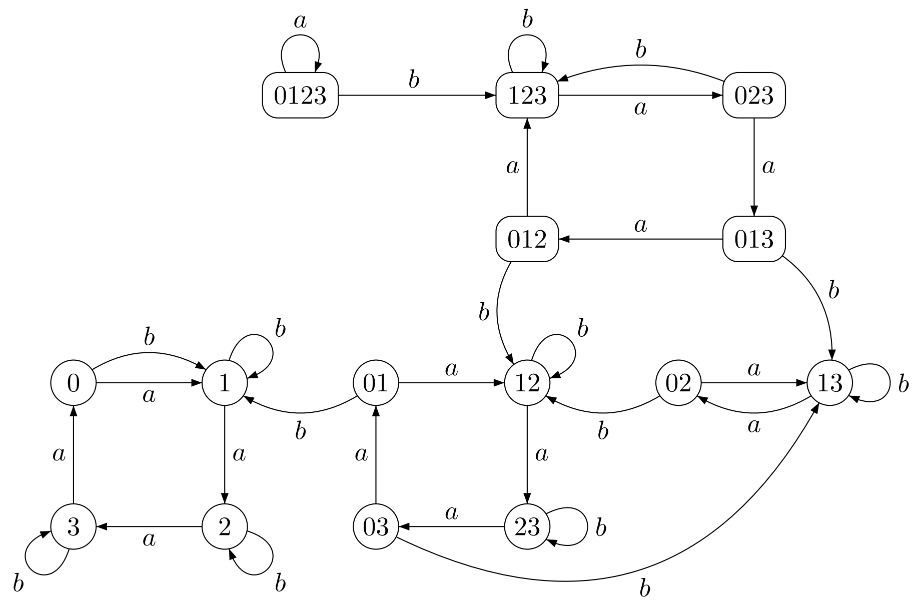

Figure 2.1: The power automaton of $\mathscr{C}_4$. The shortest reset word is `abbbaaaab`.

to check if there exists a pair X, Y of sets, belonging respectively to the BFS and IBFS lists, such that $X \subseteq Y$. Due to the *Orthogonal Vectors Conjecture* [1] [12], there is probably no subquadratic solution to this problem (we can reduce our problem to OV by transforming the sets in Y to their complements and then represent all the sets as their characteristic vectors). Nevertheless, we employ a procedure that works well in practice. This procedure is also used to reduce the number of sets on the lists during the searches, which effectively lowers the branching factor. Finally, we do not actually run the two searches until they meet. Instead, we switch to the second phase with DFS, when either the memory runs out or we calculate that it should be faster based on the upper bound from the heuristics and collected statistics.

Our improved algorithm uses the same high-level ideas, but we design it so that it is possible to solve the problem for significantly larger automata. First, we use different data structures and redesign how a single iteration of BFS / IBFS works. Apart from making the bidirectional-search phase faster, it allows it to complete more iterations before switching to the depth-first search, due to the lower memory consumption, which is crucial in case of large automata. In the DFS phase, we enhance the previously used radix trie data structure in terms of both efficiency and memory overhead. We also add some forms of list reductions, which decrease the branching factor. The decision making part of the algorithm is also improved and we use five types of steps. The profits of making a decision such as transitioning to the DFS phase or reducing the lists are now estimated using new and more advanced equations. Finally, every part of the new algorithm can be parallelized in one way or the other, which was not possible before.

[1] The *Orthogonal Vectors problem* gives two sets A, B of boolean vectors of the same length and asks if there exists a pair $(u \in A, v \in B)$ such that u and v are orthogonal, i.e., $u \cdot v = \mathbf{0}$.

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

2.2 Bidirectional breadth-first search

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

We say that the two searches *meet* if there exist $X \in L_{BFS}$ and $Y \in L_{IBFS}$ for which $X \subseteq Y$ holds. Indeed, then we know that there are words $x, y \in \Sigma^*$ such that $X = \delta(Q, x)$ and $Y = \delta^{-1}(\{q\}, y)$ for some $q \in Q$. Because $X \subseteq Y$, we get $\delta(Q, xy) = \{q\}$, which means that xy is a reset word. The meet condition implies that the lists can be reduced by removing the elements which are not minimal (and respectively maximal for IBFS) with respect to inclusion. We can reduce the lists further by ensuring that no new set is a superset (subset for IBFS) of a set belonging to some list from any previous iteration. To make this possible, we keep track of all the visited sets in two additional *history* lists H_{BFS}, H_{IBFS}. This reduction, although usually helpful during most of the iterations, at the end may turn out to be unprofitable, in which case the algorithm will drop the history list(s).

The subprocedure used to check the meet condition and reduce the lists differs between our algorithm and the original. In [15, 16], the lists were kept as dynamic *radix tries*, supporting insertion and subset (or superset) checking operations. Here, instead, we take a somewhat simpler approach and operate directly on the lists, stored as random access containers, such as vectors in C++. We call this subprocedure *MarkSupersets*(A, B) (and a similar one – *MarkProperSupersets*(A, B)). It returns which sets from B are a superset of at least one set in A. The subprocedure is described in detail in the next section.

Algorithm 1 shows the pseudocode of the bidirectional-search phase of the algorithm.

Algorithm 1 Bidirectional breadth-first search.

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

Output: Reset threshold r.

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

2.3 Subset checking

The procedure *MarkSupersets* takes lists A, B and returns a boolean list S of the same size as B, where $S[i] = \mathit{true} \iff \exists_j\ A[j] \subseteq B[i]$. The sets are treated like binary strings (their characteristic vectors) of length n and the procedure simulates building a radix trie over them on the fly. We require that A is sorted and its elements are unique. This can be guaranteed relatively cheaply and it gives us the property that every subtree in the trie built over the elements of A corresponds to a contiguous segment on the list.

Algorithm 2 Recursive procedure *MarkSupersets*

```
Input: Sorted lists $A$ and $B$ with unique elements. Current depth of recursion $d$ (0
    for the initial call).
1: if $|A| < MIN$ then                                ▷ Brute-force for small $|A|$
2:     Check each pair in $A \times B$ and update $S$.
3: end if
4: $A_0, A_1 \leftarrow A$ split by the $d$-th bit    ▷ $A$ is sorted, so a binary search suffices
5: $MarkSupersets(A_0, B)$
6: $B_1 \leftarrow$ Unmarked elements from $B$ that have $d$-th bit set to one    ▷ Linear time
   scan
7: $MarkSupersets(A_1, B_1)$
```

MarkSupersets is used to reduce the BFS lists and to check the meet condition. To implement the *MarkSubsets* procedure needed on the IBFS side, we simply convert the sets to their complements and call *MarkSupersets*.

2.4 Step cost estimation

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

We distinguish five types of steps from which the algorithm always chooses one for the next iteration – DFS, BFS, IBFS, BFS without the history list (denoted by $\overline{\text{BFS}}$) and IBFS without the history list (denoted by $\overline{\text{IBFS}}$). To assess which option to choose, we roughly estimate the number of subset checking operations each of them will require. We reuse some of the equations used in [16]. Using simplifying assumptions about the uniform distribution of the states in sets and their *ExpNvn* equation, we bound the expected number of subset checking operations in a call to *MarkSupersets* with lists of sizes A_s, B_s and their densities:

$$density(A) = \frac{\sum_{Q \in A} |Q|}{n|A|}$$

equal to A_d, B_d, with an equation

$$\begin{aligned} &ExpMark(A_s, B_s, A_d, B_d) = \\ &\qquad = B_s\Big(\frac{1+B_d}{B_d} + \frac{1}{A_d - A_d B_d}\Big) A_s^{\log_w(1+B_d)}, \end{aligned}$$

where

$$w = \frac{1 + B_d}{1 + A_d B_d - A_d}.$$

To estimate the sizes of the lists after (and in between) the reductions, we store the ratio r_t of reduced sets in the previous iterations for each type of reduction t. We use the following equations to calculate the expected costs of single iterations of the bidirectional-search options.

$$\begin{aligned} BFS_{cost} = {} & ExpMark(K \cdot (1 - r_1) \cdot |L_{BFS}|, \quad K \cdot (1 - r_1) \cdot |L_{BFS}|, \\ & \qquad density(L_{BFS}), \quad density(L_{BFS})) \\ & + ExpMark(|H_{BFS}|, \quad K \cdot (1 - r_1) \cdot (1 - r2) \cdot |L_{BFS}|, \\ & \qquad density(H_{BFS}), \quad density(L_{BFS})) \\ & + ExpMark(K \cdot (1 - r_1) \cdot (1 - r_2) \cdot (1 - r_3) \cdot |L_{BFS}|, \quad |L_{IBFS}|, \\ & \qquad density(L_{BFS}), \quad density(L_{IBFS})), \end{aligned}$$

where r_1, r_2, r_3 are the reduced sets ratios of respectively the removal of duplicates, the removal of non-minimal subsets and the reduction by history list.

$$\begin{aligned} \overline{BFS_{cost}} = {} & ExpMark(K \cdot (1 - r_1) \cdot |L_{BFS}|, \quad K \cdot (1 - r_1) \cdot |L_{BFS}|, \\ & \qquad density(L_{BFS}), \quad density(L_{BFS})) \\ & + ExpMark(K \cdot (1 - r_1) \cdot (1 - r_2) \cdot |L_{BFS}|, \quad |L_{IBFS}|, \\ & \qquad density(L_{BFS}), \quad density(L_{IBFS})), \end{aligned}$$

where r_1, r_2 are the reduced sets ratios of respectively the removal of duplicates and the removal of non-minimal subsets.

$$\begin{aligned} IBFS_{cost} = {} & ExpMark(K \cdot (1 - r_1) \cdot |L_{IBFS}|, \quad K \cdot (1 - r_1) \cdot |L_{IBFS}|, \\ & \qquad 1 - density(L_{IBFS}), \quad 1 - density(L_{IBFS})) \\ & + ExpMark(|H_{IBFS}|, \quad K \cdot (1 - r_1) \cdot (1 - r2) \cdot |L_{IBFS}|, \\ & \qquad 1 - density(H_{IBFS}), \quad 1 - density(L_{IBFS})) \\ & + ExpMark(|L_{BFS}|, \quad K \cdot (1 - r_1) \cdot (1 - r_2) \cdot (1 - r_3) \cdot |L_{IBFS}|, \\ & \qquad density(L_{BFS}), density(L_{IBFS})), \end{aligned}$$

for r_1, r_2, r_3 same as in BFS_{cost}.

$$\begin{aligned} \overline{IBFS_{cost}} = {} & ExpMark(K \cdot (1 - r_1) \cdot |L_{IBFS}|, \quad K \cdot (1 - r_1) \cdot |L_{IBFS}|, \\ & \qquad 1 - density(L_{IBFS}), \quad 1 - density(L_{IBFS})) \\ & + ExpMark(|L_{BFS}|, \quad K \cdot (1 - r_1) \cdot (1 - r_2) \cdot |L_{IBFS}|, \\ & \qquad density(L_{BFS}), \quad density(L_{IBFS})), \end{aligned}$$

for r_1, r_2 same as in $\overline{BFS_{cost}}$.

Additionally, if we cannot perform some step because there is not enough memory, we set its expected cost to ∞. Once we choose $\overline{\text{BFS}}$, we no longer consider the BFS step, so in this case we also set its cost to ∞ (this is symmetrical for $\overline{\text{IBFS}}$ and IBFS).

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

$$F = K \cdot (1 - r_1),$$

where r_1 is the ratio of duplicates removed in the IBFS list during the latest reduction (since the DFS also removes duplicates). We assume that the reset threshold is equal to the known upper bound and from that, we get the expected number of iterations that still need to be done, which we denote by I. The predicted full costs of each option are as follows:

$$\begin{aligned}
DFS_{pred} &= F \cdot \frac{F^I - 1}{F - 1} \\
&\quad \cdot ExpMark(|L_{BFS}|,\ |L_{IBFS}|, \\
&\quad density(L_{BFS}),\ density(L_{IBFS})) \\
BFS_{pred} &= BFS_{cost} + F \cdot \frac{F^{(I-1)} - 1}{F - 1} \\
&\quad \cdot ExpMark(K \cdot (1 - r_1) \cdot (1 - r_2) \cdot (1 - r_3) \cdot |L_{BFS}|,\ |L_{IBFS}|, \\
&\quad density(L_{BFS}),\ density(L_{IBFS})) \\
\overline{BFS_{pred}} &= \overline{BFS_{cost}} + F \cdot \frac{F^{(I-1)} - 1}{F - 1} \\
&\quad \cdot ExpMark(K \cdot (1 - r_1) \cdot (1 - r_2) \cdot |L_{BFS}|,\ |L_{IBFS}|, \\
&\quad density(L_{BFS}),\ density(L_{IBFS})) \\
IBFS_{pred} &= IBFS_{cost} + \frac{F^{(I-1)} - 1}{F - 1} \\
&\quad \cdot ExpMark(|L_{BFS}|,\ K \cdot (1 - r_1) \cdot (1 - r_2) \cdot (1 - r_3) \cdot |L_{IBFS}|, \\
&\quad density(L_{BFS}),\ density(L_{IBFS})) \\
\overline{IBFS_{pred}} &= \overline{IBFS_{cost}} + F \cdot \frac{F^{(I-1)} - 1}{F - 1} \\
&\quad \cdot ExpMark(|L_{BFS}|,\ K \cdot (1 - r_1) \cdot (1 - r_2) \cdot |L_{IBFS}|, \\
&\quad density(L_{BFS}),\ density(L_{IBFS}))
\end{aligned}$$

Finally, the step with the lowest predicted full cost is chosen. As an exception to that rule, if we do not expect to switch into the DFS phase soon, instead of comparing the prediction costs of every choice, we consider only the BFS and IBFS costs. In this case, we greedily choose the one with the lower single-step cost, which makes the bidirectional-search more balanced and faster in the short term (as otherwise, BFS is strongly preferred due to the assumption that the rest of the steps will be DFS).

2.5 Depth-first search

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

2.5.1 Static radix trie

The L_{BFS} list is stored in an optimized data structure that supports the *ContainsSubset* operation. It is based on a radix trie, in which the characteristic vectors of the sets are stored. The queries rely on traversing the tree and, in each step, descending to either both children or just the left (zero) child, depending on the queried set. In this way, we can omit checking the queried set with many sets stored in the trie, especially if its cardinality is low. Figure 2.2 shows an example.

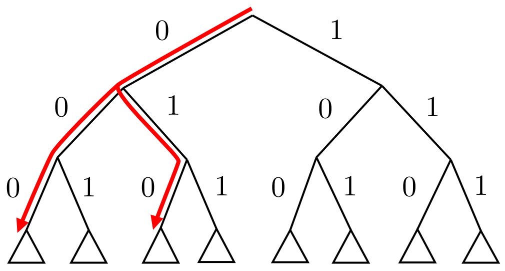

Figure 2.2: Possible traversal of the radix trie during a call to *ContainsSubset*(*010*...)

Our trie can be built in time $\mathcal{O}(|L_{BFS}|\,N^2)$, and it contains the following enhancements that additionally speed up the queries.

Dynamic state ordering

In each node at a depth d, instead of splitting the subtrees by the d-th state as in a regular radix trie, we split by a state x chosen specifically for the node. The state x is chosen so that the number of sets stored in the current subtree that also contain x

is the largest possible. In practice, this makes the queries faster, because in vertices such that x does not belong to the queried set, a large number of sets is immediately skipped.

This optimization also implies path compression, as we do not have nodes which do not split the sets into two non-empty parts. To ensure this, we also exclude the case that x is contained in all the sets and do not use it as a division state.

Leaf threshold

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

Mask and size elimination

Every node v stores the intersection of the sets contained in its subtree U_v and the minimal size of a set in this subtree m_v. When we query for subsets of a set X, if $U_v \subseteq X$ or $m_v \leq |X|$ does not hold, we do not recurse into the subtree of v.

This enhancement was also used in the original algorithm, but without the *Leaf threshold* optimization the memory overhead was larger and therefore sometimes keeping U_v was disadvantageous, especially for large automata.

2.5.2 DFS

During the search, at each depth the current list is split into parts of size at most

$$\frac{available\_memory}{(k+1)(upper\_bound - lower\_bound)}$$

and recursed into one by one, to make sure we do not run out of memory. The elements are sorted in order of descending cardinality, so that the most promising sets are recursed into in the first place, which in turn can quickly improve the upper bound, if it was not tight. The lists are reduced by removal of duplicates and calls to *MarkSubsets* only once every few iterations, which still lowers the branching factor significantly.

Algorithm 3 Recursive procedure *DFS*.

Input: Static radix trie T built over the elements of L_{BFS}. List L – initially L_{IBFS}. The current reset word length r (BFS iterations + recursion depth). The reset threshold upper bound R, which can be changed globally once we find a shorter reset word.

Output: R will be set to match the reset threshold.

```
 1: $L_{next} \leftarrow CalculatePreimages(L)$
 2: Sort $L_{next}$ in the order of decreasing cardinality
 3: if Time for duplicates removal then
 4:     Remove duplicates from $L_{next}$
 5: end if
 6: if Time for reduction then
 7:     Delete subsets of the $C$ largest sets in $L_{next}$ ($MarkSubsets$)  ▷ $C$ – parameter
 8: end if
 9: for $s \in L_{next}$ do
10:     if $T$ contains subset of $s$ then
11:         $R \leftarrow r$
12:         return
13:     end if
14: end for
15: if $r = R - 1$ then
16:     return                      ▷ Failed to find a reset word shorter than $R$
17: end if
18: for $L_{part} \in Split(L_{next})$ do        ▷ Split into smaller parts if necessary
19:     $DFS(T, L_{part}, r + 1, R)$
20:     if $r = R - 1$ then
21:         return
22:     end if
23: end for
```

Chapter 3

Implementation

We implemented the exact algorithm described in the previous chapter along with several known heuristic algorithms. The code is written in C++17 and should work on most Linux distributions. It is available for download at

`https://github.com/marekesz/synchrowords`

The thesis is related to Release 1.0.0.

The implementation is intended to be as efficient as possible and in the next sections, we describe some of the techniques we used in order to achieve this.

3.1 Hot spots

Below we list the bottlenecks of our exact algorithm. We tried to optimize them in terms of execution time and memory footprint. Note that because we use just-in-time compilation, parameters such as n and k are known during the compilation of the following code.

3.1.1 The subset data structure

The most widely used data structure is undoubtedly `Subset<S>` (Listing 3.1). All of the implemented algorithms (though for different reasons) store subsets of states of the automaton. We implement this structure as a bitset using an array of integers. We further assume that the states of the automaton are numbered from 0 to $n-1$ and they index bits in the subset structure.

```
template <uint S>
struct Subset {
  uint64 v[buckets(S)];
  ... // Member functions
};
```

Listing 3.1: `Subset<S>` in `synchrolib/data_structures/subset.hpp`.

Inside some of its methods we also use `__builtin` functions available in *gcc* compiler. One such case is the use of `__builtin_ctzll` inside functions that need to iterate over the set bits quickly.

3.1.2 Computing images and preimages of subsets

Every pathfinding-based algorithm needs a subprocedure to identify the successors of a given state in the power automaton. Because the states are stored as bitsets, we want to iterate over these bits and for each of them, set some bits in the output, according to the transition function of the original automaton. In the case of computing images under some $w \in \Sigma$, a straightforward implementation is quite efficient, as each set bit b in the original bitset will require exactly one operation on the output bitset – setting the $\delta(b, w)$-th bit. For the preimages it is less efficient, as each set bit b requires setting $|\delta^{-1}(b, w)|$ bits in the output, which means a nested loop would be needed. For this reason, we choose a different approach for both image and preimage computation, based on cutting the bitset into small chunks and merging the precomputed outputs (Listing 3.2). This works well in practice, as there is no branching and the compiler can unroll all of the loops easily.

```
template <uint N, uint K>
struct PreprocessedTransition {
  constexpr static uint SLIZE_SIZE = 8;
  constexpr static uint SUBSETS_BITS = 64;
  Subset<N> trans[slices()][(1 << SLICE_SIZE)];

  void apply(const Subset<N> &from, Subset<N> &s) const {
    for (uint b = 0; b < s.buckets(); b++) s.v[b] = 0;
    for (uint i = 0; i < slices(); i++) {
      uint b = i * SLICE_SIZE / SUBSETS_BITS;
      uint shift = i * SLICE_SIZE % SUBSETS_BITS;
      uint set = (from.v[b] >> shift) & ((1 << SLICE_SIZE) - 1);
      s |= trans[i][set];
    }
  }
  ...
};
```

Listing 3.2: `PreprocessedTransition<N, K>` in `synchrolib/data_structures/preprocessed_transition.hpp`.

In most cases, we want to compute images or preimages of a lot of sets. We can easily parallelize this process by distributing them to different (CPU or GPU) threads. For the CUDA implementation see:
`synchrolib/data_structures/cuda/preprocessed_transition_kernel.cu`.

3.1.3 MarkSupersets

The *MarkSupersets*(A, B) function (Algorithm 2) is the undisputed bottleneck of the bidirectional-search phase in our exact algorithm. It is used both to check if we have already found the reset threshold (meet criterion) and to reduce the sizes of the lists by removing unnecessary sets. We implement it as a recursive function that takes ranges of A and B as arguments. It means that the function will have to change the order of elements in B, in order to place some of them in a contiguous range before the recursive call. We show a simplified (and slightly modified) implementation of this function in Listing 3.3.

```
void contains_subset(uint depth, Iterator A_begin, Iterator A_end,
    Iterator B_begin, Iterator B_end) {
  if (std::distance(A_begin, A_end) <= MIN) {
    if constexpr (!GPU) {
      // check each pair from A × B and update answers
    } else {
      // do the same but as a CUDA kernel
    }
    return;
  }

  Iterator A_mid = binsearch_first_one(depth, A_begin, A_end);
  contains_subset(depth + 1, A_begin, A_mid, B_begin, B_end);

  Iterator B_mid = shift_ones_left(depth, B_begin, B_end);
  contains_subset(depth + 1, A_mid, A_end, B_begin, B_mid);
}
```

Listing 3.3: Simplified `SubsetsImplicitTrie::check_contains_subset_impl` in `synchrolib/data_structures/subsets_implicit_trie.hpp`.

We can easily parallelize this function on the CPU by distributing the list B to different threads. It is not possible to do the same thing on GPU, because the recursive nature of the procedure requires a lot of branching. Instead, we increase the *MIN* constant and run the brute-force part as a CUDA kernel. This creates a trade-off between the amount of computation that the (slower) CPU has to do and the number of pairs that the (faster) GPU has to check. The optimal value of *MIN* greatly depends on the hardware. In our experiments, we used $MIN = 10^4$.

3.1.4 Static radix trie

When the algorithm switches to the depth-first search phase, a radix trie is built on the elements of the BFS list (Listing 3.4).

```
void build(uint v, Iterator begin, Iterator end) {
  if (std::distance(begin, end) <= MIN) {
    // store the iterators, as well as AND and minimum size of the sets
    nodes[v].store(begin, end);
    return;
  }
  // get the bit with the most (but not all) ones
  uint division_bit = get_division_bit(begin, end);
  nodes[v].division_bit = division_bit;
  Iterator mid = shift_zeros_left(division_bit, begin, end);

  if (begin != mid) {
    nodes[v].zero = create_node();
    // build the subtree
    build(nodes[v].zero, begin, mid);
    // update subtree AND and minimum set size
    nodes[v].update(nodes[nodes[v].zero]);
  }
  if (mid != end) {
    nodes[v].one = create_node();
    build(nodes[v].one, mid, end);
    nodes[v].update(nodes[nodes[v].one]);
  }
}
```

Listing 3.4: Simplified `SubsetsTrie::build_impl_swap` in `synchrolib/data_structures/subsets_trie.hpp`.

To find the most common bit, `get_division_bit` function has to scan the whole `<begin, end)` range and for each subset iterate over its set bits. Fortunately, this is not a big concern for us, because the trie is built only once. Moreover, the number of states in each of the stored sets is usually small. This is caused by the fact that the subsets come from the BFS list, which made roughly $\frac{r}{2}$ iterations and we expect $|\delta(Q, w)|$ to be small for $|w| \geq \frac{r}{2}$. In our parallel implementation, we divide the range for multiple threads in `get_division_bit`.

To check the meet condition during the DFS, we call `contains_subset`(Listing 3.5) for each of the sets in the list. We can do this on multiple CPU threads.

```
bool contains_subset(const Node& node, const Subset<N>& set) const {
  if (set.size() < node.subtree_min_popcount) return false;
  if (!set.is_subset(node.subtree_and)) return false;
  if (set.is_subset(node.subsets_range)) return true;
  if (node.division_bit == N) return false;

  if (!set.is_set(node.division_bit)) {
    return node.zero && contains_subset(nodes[node.zero], set);
  }

  if (node.zero && contains_subset(nodes[node.zero], set)) {
    return true;
  }
  return node.one && contains_subset(nodes[node.one], set);
}
```

Listing 3.5: Simplified `SubsetsTrie::contains_subset_of_impl` in `synchrolib/data_structures/subsets_trie.hpp`.

3.1.5 Sorting and removal of duplicates

Sorting and removing duplicate subsets take a significant amount of time in the DFS phase. To make the algorithm faster, we do these only once every two iterations and we (optionally) sort using multiple CPU threads [1] (Listing 3.6).

```
template<class T>
void sort(T* data, int len, int grainsize) {
  if (len < grainsize) {
    std::sort(data, data + len, std::less<T>());
  } else {
    auto future = std::async(sort<T>, data, len / 2, grainsize);
    sort<T>(data + len / 2, len - len / 2, grainsize);
    future.wait();
    std::inplace_merge(data, data + len / 2, data + len, std::less<T
      >());
  }
}
```

Listing 3.6: `parallel_sort` in `synchrolib/utils/vector.hpp`.

[1] Code copied from `https://codereview.stackexchange.com/questions/22744/multi-threaded-sort`

3.2 Just-in-time compilation

To be able to generate a well-optimized, memory-efficient code, it is crucial that the compiler knows the number of states and the size of the alphabet during its work. It also helps to have the algorithm's parameters as constants. Because we do not want to force the user to manually recompile the program for each size of the input, we provide a small just-in-time (JIT) compilation library `jitlib.hpp`.

JitLib works by making a substitution in the code according to the specified $(\mathit{pattern}, \mathit{substitution})$ pairs. Then the files are compiled into a dynamic library, linked to the main program via `dlopen(3)`, and finally the targeted function found by `dlsym(3)` is run (Listing 3.7 and Listing 3.8).

```cpp
template <typename... Ts, typename... Args>
JitLib& run(std::string func_name, Args&&... args) {
  Logger() << "Finding function " << func_name;
  dlerror(); // clear errors
  *(void**)&func_ = dlsym(dl_, func_name.c_str());
  auto err = dlerror();
  if (err || !func_) {
    Logger() << "Could not find function " << func_name << ":\n"
        << err;
    throw RunException("function not found");
  }

  Logger() << "Running function";
  reinterpret_cast<void (*)(Ts...)>(func_)(std::forward<Args>(args)
      ...);
  return *this;
}
```

Listing 3.7: JitLib::run from `jitlib/jitlib.hpp`.

```cpp
int a = 1;
int b = 2;
int c;
jitlib
    .substitute(sources_paths, dest_path, substitutions_map)
    .compile()
    .load("libexample.so")
    .run<int, int, int&>("add_two_ints", a, b, c)
    .run<int>("print_int", c);
```

Listing 3.8: Example usage of JitLib.

In our program, we reuse the previously compiled libraries, so that for one config (see Appendix: User guide), the number of states, and the size of the alphabet, the algorithm(s) is compiled only once.

As an example benefit of knowing the number of states at compile time, Listing 3.9 and Listing 3.10 show how the `gcc` compiler unrolls the loop in the `|=` operator in our `Subset<S>` structure for $S = 500$.

```
template <uint S>
struct Subset {
  uint64 v[buckets(S)];

  Subset<S>& operator|=(const Subset<S>& s) {
    for (uint b = 0; b < buckets(S); b++) {
      v[b] |= s.v[b];
    }
    return *this;
  }
};
```

Listing 3.9: `Subset<S>::operator |=` in C++.

```
movdqu (%rsi), %xmm1
movdqu (%rdi), %xmm0
movdqu 48(%rdi), %xmm4
por %xmm1, %xmm0
movups %xmm0, (%rdi)
movdqu 16(%rdi), %xmm0
movdqu 16(%rsi), %xmm2
por %xmm2, %xmm0
movups %xmm0, 16(%rdi)
movdqu 32(%rdi), %xmm0
movdqu 32(%rsi), %xmm3
por %xmm3, %xmm0
movups %xmm0, 32(%rdi)
movdqu 48(%rsi), %xmm0
por %xmm4, %xmm0
movups %xmm0, 48(%rdi)
```

Listing 3.10: `Subset<500>::operator |=` in Assembly (x86-64 gcc 11.2, -O3).

3.3 Computational plans

In our implementation, we use `json` computational plans for specifying which algorithms should be run. A computational plan consists of global parameters (such as the allowed number of threads), a list of algorithms to be run, and, optionally, some algorithm-specific parameters. Listing 3.11 shows an example computational plan (the meaning of the specific parameters is described in Appendix: User guide).

```
{
    "upper_bound": "1ULL * AUT_N * AUT_N * AUT_N / 6",
    "threads": 1,
    "gpu": false,
    "algorithms": [
        {
            "name": "Beam",
            "config": {
                "presort": "indeg",
                "beam_size": "std::log2(AUT_N) * AUT_N * 3"
            }
        },
        {
            "name": "Exact",
            "config": {
                "dfs_min_list_size": "10000",
                "bfs_small_list_size": "AUT_N * 16",
                "dfs": true,
                "dfs_shortcut": true,
                "max_memory_mb": "7 * 1024"
            }
        }
    ]
}
```

Listing 3.11: Example computational plan.

The algorithms are run sequentially in the given order and each of them can modify the overall result. A typical goal of each algorithm is to lower the upper bound on the reset threshold found by its predecessors. It is not always the case though, as the algorithms can also share different data between each other. As an example, the goal of the `Reduce` algorithm is to lower the number of states of the automaton before the `Exact` algorithm (by keeping only the states which are inside the sink component [15, 16]).

```
{
    "algorithms": [
        {
            "name": "Beam",
            "config": {...}
        },
        {
            "name": "Reduce",
            "config": {
                "min_n": "80",
                "list_size_threshold": "AUT_N * 16"
            }
        },
        {
            "name": "Exact",
            "config": {...}
        }
    ]
}
```

Listing 3.12: A computational plan with the `Reduce` algorithm

3.4 Adding new algorithms

It is easy to integrate other algorithms with the application and create new computational plans based on them. To do so, one needs to:

1. Create a new directory in `synchrolib/algorithm` and add (at least) two header files – one for the implementation of the algorithm and one for the config definition. The implementation will be compiled using our just-in-time compilation library (section 3.2).

2. Implement the algorithm as a class derived from `Algorithm<N, K>` (see Listing 3.13).

```
template <uint N, uint K>
class Algorithm {
public:
  virtual void run(AlgoData<N, K>& data) = 0;
  virtual ~Algorithm() = default;
};
```

Listing 3.13: `Algorithm<N, K>` in `synchrolib/algorithm/algorithm.hpp`.

The `AlgoData<N, K>` class contains the input automaton `Automaton<N, K>` and the data generated by the algorithms, stored as `AlgoResult` (Listing 3.14). The same instance is passed to each run in a computational plan, so if there is a need for data transfer between the algorithms, it can be achieved by adding new fields to `AlgoResult` (preferably of type `std::optional<T>`).

```
struct AlgoResult {
  bool non_synchro;
  uint64 mlsw_lower_bound;
  uint64 mlsw_upper_bound;
  std::optional<FastVector<uint>> word;

  size_t algorithms_run;

  std::optional<ReduceData> reduce; // field added for data transfer between
      Reduce and Exact algorithms
  ...
};

template <uint N, uint K>
struct AlgoData {
  const Automaton<N, K> aut;
  const InverseAutomaton<N, K> invaut;

  AlgoResult result;
  ...
};
```

Listing 3.14: `AlgoResult` and `AlgoData<N, K>` in `synchrolib/algorithm/algorithm.hpp`

The implementation can make use of constants known at compilation time. Common constants such as `AUT_N`, `AUT_K`, `GPU` and `THREADS`, are defined in `jitdefines.hpp`, which can be included in just-in-time compiled files. Algorithm-specific constants can be added by defining *substitutions* inside the algorithms config class (this is described in the next step).

3. Create a config definition – a class derived from `AlgoConfig` (Listing 3.15).

```
class AlgoConfig {
public:
  virtual std::vector<std::pair<std::string, std::string>>
  get_substs(const json& config) const = 0;
};
```

Listing 3.15: `AlgoConfig` in `synchrolib/algorithm/config.hpp`

The `get_subst` function is called with the contents of the algorithm specific configuration provided by the user. It returns (*pattern*, *substitution*) pairs, which are then used to change the implementation code before its compilation. As an example, let us consider a `Dummy` algorithm, which sets the upper bound on the reset threshold to a value named `constant` from its json config (Listing 3.16).

```
class DummyConfig : public AlgoConfig {
public:
  std::vector<std::pair<std::string, std::string>> get_substs(
      const json& config) const override {
    return {
      {"$DUMMY_DEF$", def(config)},
      {"$DUMMY_UNDEF$", undef()}
    };
  }

private:
  static std::string def(const json& config) {
    return make_define("CONSTANT", config.value("constant", 0));
  }

  static std::string undef() {
    return make_undefine("CONSTANT");
  }
};
```

Listing 3.16: Example `DummyConfig`.

The patterns `$DUMMY_DEF$` and `$DUMMY_UNDEF$` in the algorithm implementation files will be substituted respectively to `#define CONSTANT <value from config>` and `#undef CONSTANT`. Notice the use of the `make_define` and `make_undefine` functions from `synchrolib/algorithm/config.hpp`. There are more similar config-related helper functions in this file. Now we are ready to implement the algorithm: Listing 3.17.

```
#include <jitdefines.hpp> // included to use AUT_N

$DUMMY_DEF$ // substituted to #define CONSTANT <constant>

template <uint N, uint K>
class Dummy : public Algorithm<N, K> {
```

```
public:
  void run(AlgoData<N, K>& data) override {
    static_assert(CONSTANT > 0, "constant must be positive");
    static_assert(CONSTANT < 1ll * AUT_N * AUT_N * AUT_N, "constant
        must be lower than N^3");
    data.result.mlsw_upper_bound = CONSTANT;
  }
};

$DUMMY_UNDEF$ // substituted to #undef CONSTANT
%
```

Listing 3.17: Example `Dummy<N, K>` implementation

4. After implementing the algorithm and defining the config, one needs to enable it in the application. To do that, a few boilerplate lines must be added inside `synchrolib/synchrolib.hpp`, which contains instructions on what to do exactly.

Chapter 4

Experiments

4.1 Setup

4.1.1 Hardware and software

The experiments were run on the computational grid in the Institute of Computer Science, University of Wrocław (funded by National Science Centre, Poland, under project number 2019/35/B/ST6/04379). The used computers were equipped with AMD Ryzen Threadripper 3960X 24-Core Processor, 64GB RAM, and two RTX3080 Nvidia GPU cards.

On the software side, we compiled the code using `gcc 9.3.0` and `nvcc 10.1` (run with `gcc 7.5.0`). To generate the tests and execute the algorithms we used `python3` and `bash` scripts.

4.1.2 The setup

We test our implementation for random binary automata. A *random automaton* (Q, Σ, δ) with n states and k letters is generated by choosing the state for each transition $\delta(q, a)$, for $q \in Q$, $a \in \Sigma$, uniformly at random from Q. Most of automata are synchronizing [3, 19], thus we can easily obtain any desired number of synchronizing ones for experiments. Concerning the presented results, we did not encounter any non-synchronizing automaton in any sample.

For the time results, compilation time is not included, as it could slightly distort the results since compilation happens just once for each n. Additionally, we do not include the execution times of the polynomial heuristic algorithms, which are negligible for the hardest cases (also, we believe they can be highly optimized).

4.2 Comparison with the previous algorithm

We call the previous algorithm *old* [16] and use its original implementation. For our new algorithm, we use the configuration plan `exact.json` and set the number of threads, memory for the exact algorithm, and GPU usage as specified. In both algorithms, we do not use reduction of the automaton, which requires recompilation.

Figure 4.1 shows the comparison of the efficiency of the old algorithm and the new one in four different configurations that differ by computational resources. This is shown in terms of reset thresholds, as it is the main parameter of an automaton that influences the running time. Figure 4.2 shows the same results in a logarithmic scale. Figure 4.3 shows how the running time increases as the reset threshold grows. The lower ratio of our algorithm is clearly visible and is also reflected on Figure 4.4, where we plot the average speedup compared to the old algorithm.

Since for a given number of states n we obtain random automata with a reset threshold from a relatively large range, averaging running time for an n has a high variance, unless very large samples are used. Figure 4.5 shows the results in terms of the number of states. The old algorithm did not manage to solve the 70 instances in time for $n > 370$. In fact, for $n = 400$ it solved only five cases in the given 20 hours (the sixth automaton had reset threshold equal to 57), while our fastest configuration needed less than 2.5 hours to complete the full 70-case test.

4.3 Testing large automata

For automata larger than 400, we use our strongest configuration plan `exact_reduce.json`, where we set the maximum memory to 30GiB and use 6 threads together with GPU. We also modified the beam size to $6n^2$.

For each $n \in [410, 420, \dots, 500]$ we sampled 40 binary random automata. Figure 4.6 shows the correspondence between the average running time and the reset threshold. Figure 4.7 shows this correspondence between n instead, which has a high variation due to small samples. Finally, Figure 4.8 shows the average running time growth depending on the reset threshold. Together with Figure 4.6, this shows that the running time growth rate for large automata is stable in terms of the reset threshold: increasing the reset threshold by 1 requires ~ 1.5 times more computation time. The same trend was visible for medium automata in Figure 4.3.

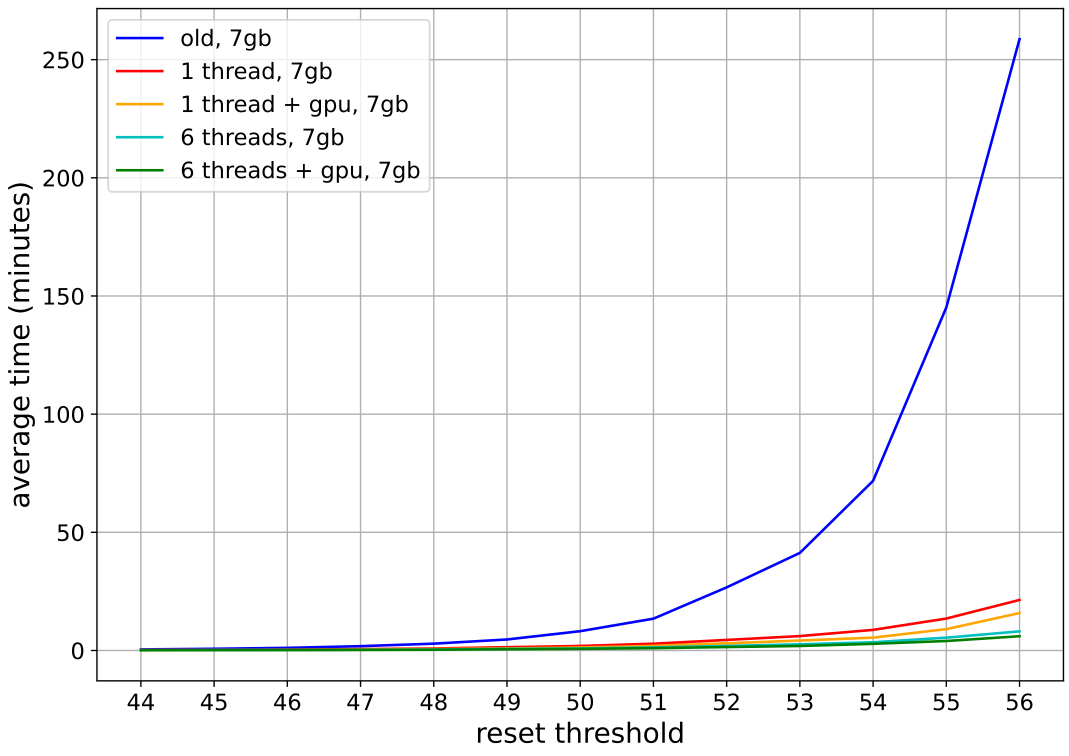

Figure 4.1: The average running time for reset thresholds above 44 that appeared for sampled 70 random binary automata for each $n \in [100, 110, \ldots, 360, 370]$.

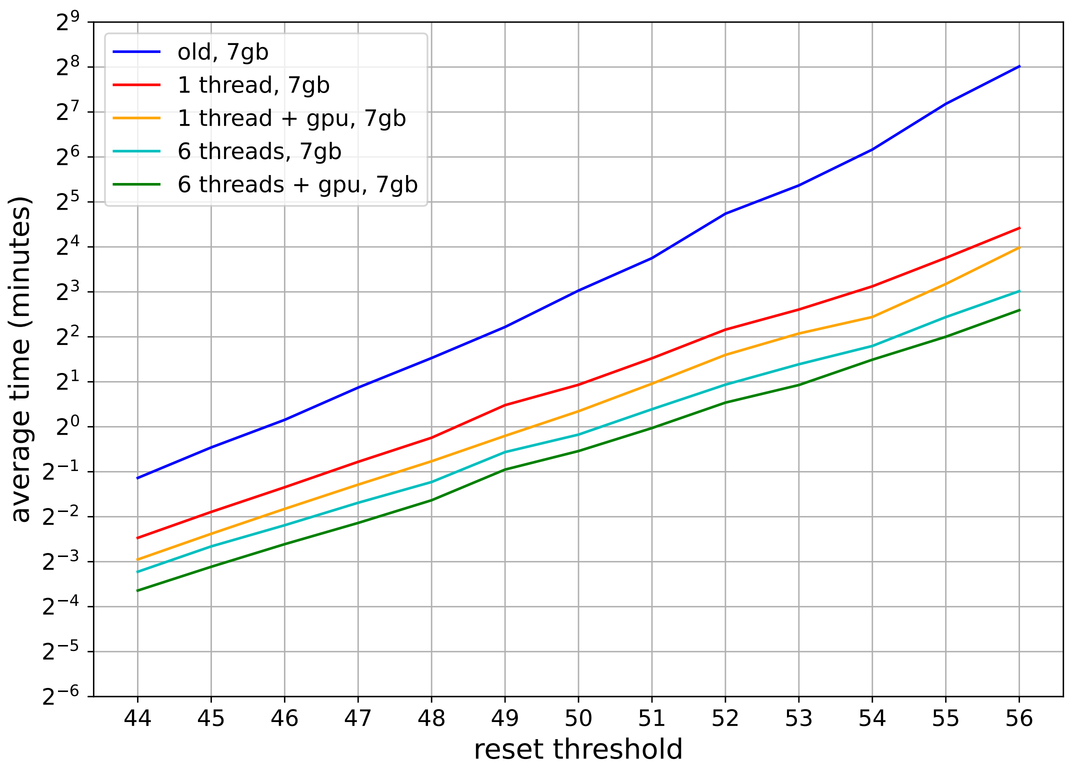

Figure 4.2: The average running time for medium reset thresholds in a logarithmic scale (the same results for larger automata are in Figure 4.6).

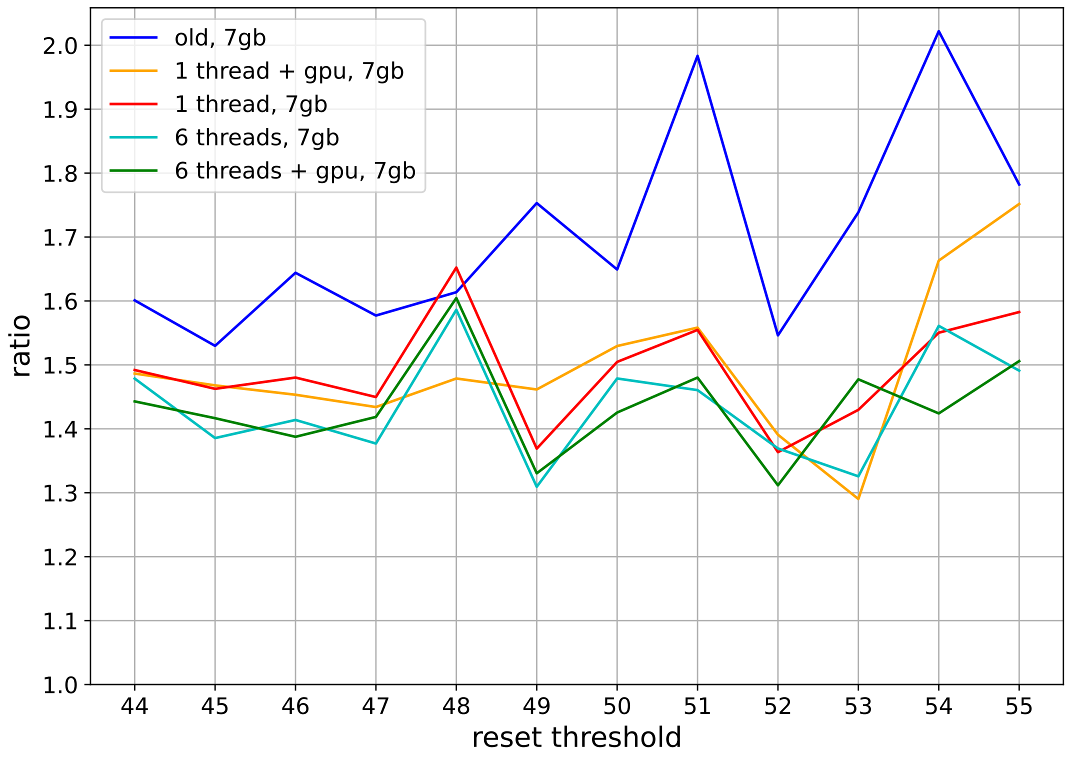

Figure 4.3: The ratio between average times for reset thresholds $x + 1$ and x.

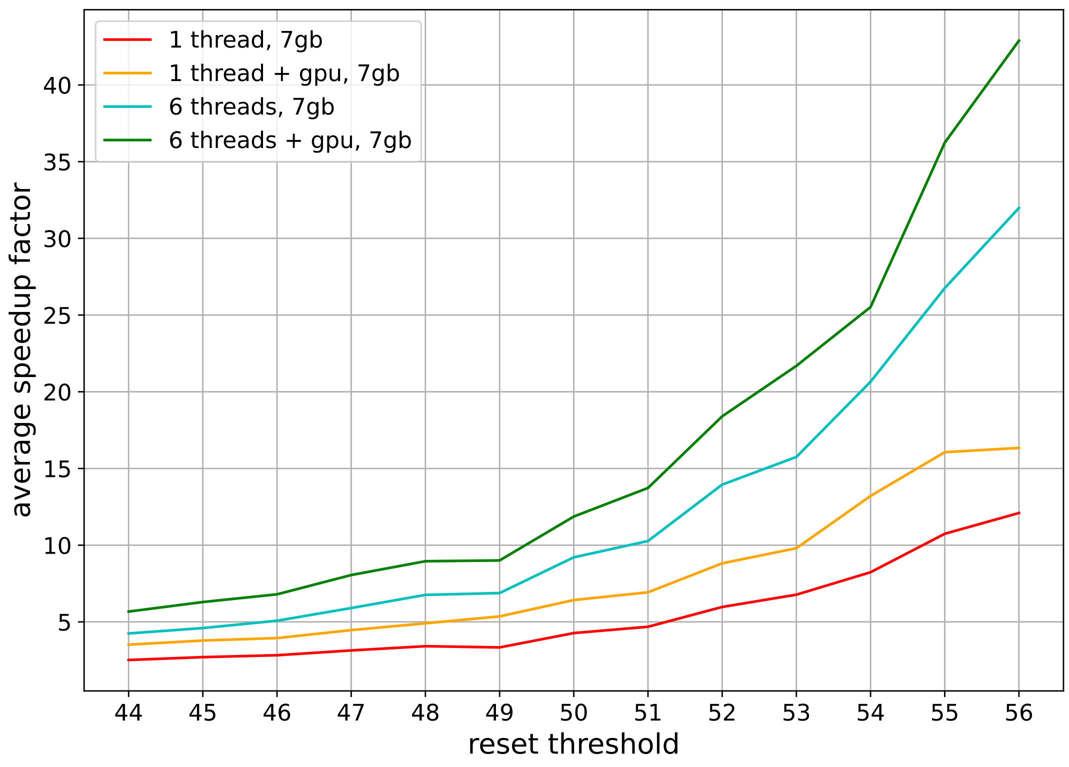

Figure 4.4: The average speed-up factor with respect to the old algorithm for medium reset thresholds.

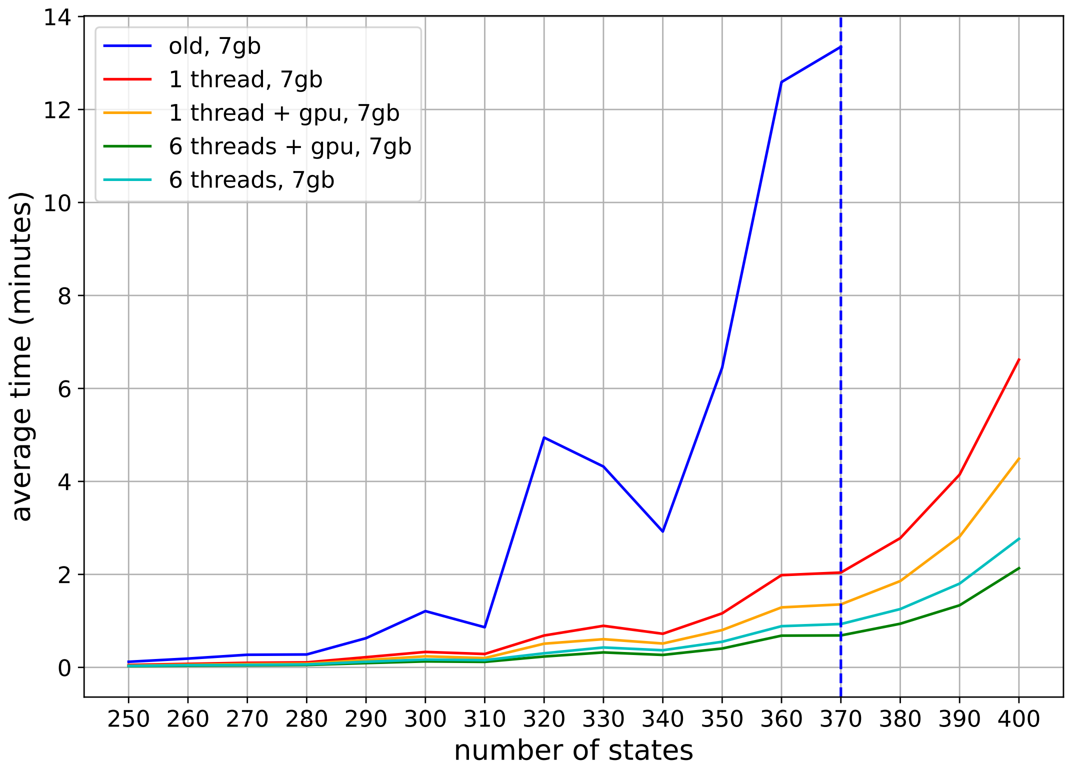

Figure 4.5: The average running times for automata with up to 400 states.

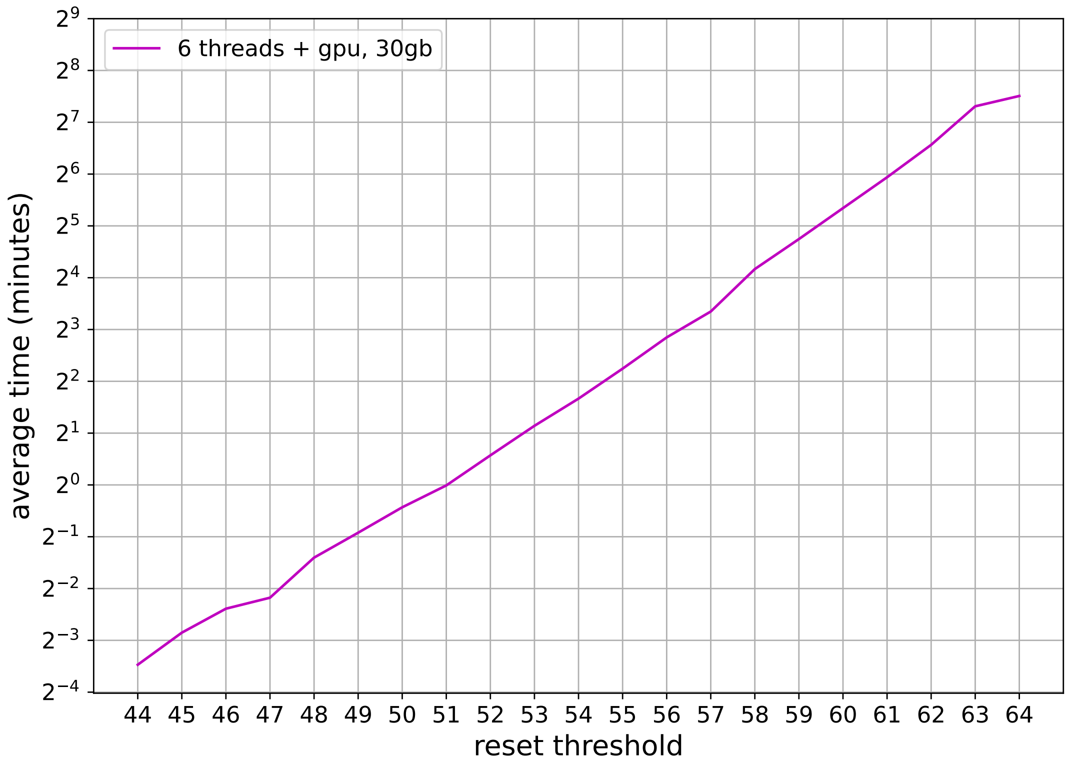

Figure 4.6: The average running times for large reset thresholds in a logarithmic scale.

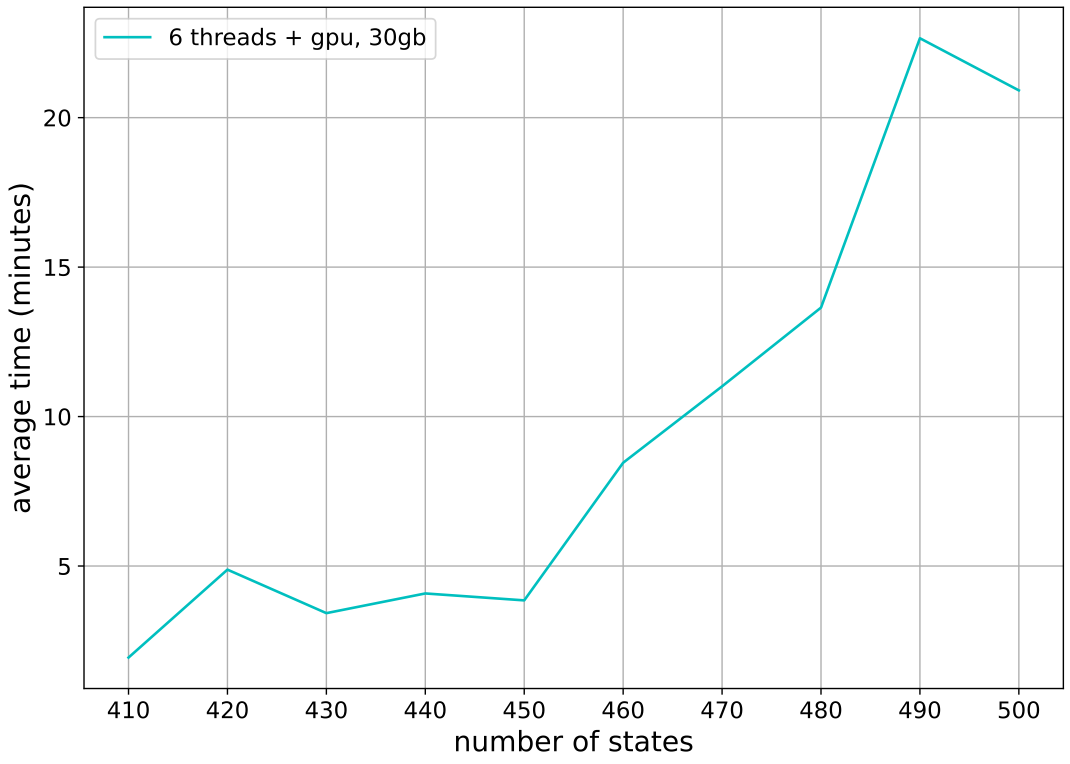

Figure 4.7: The average running times for large number of states.

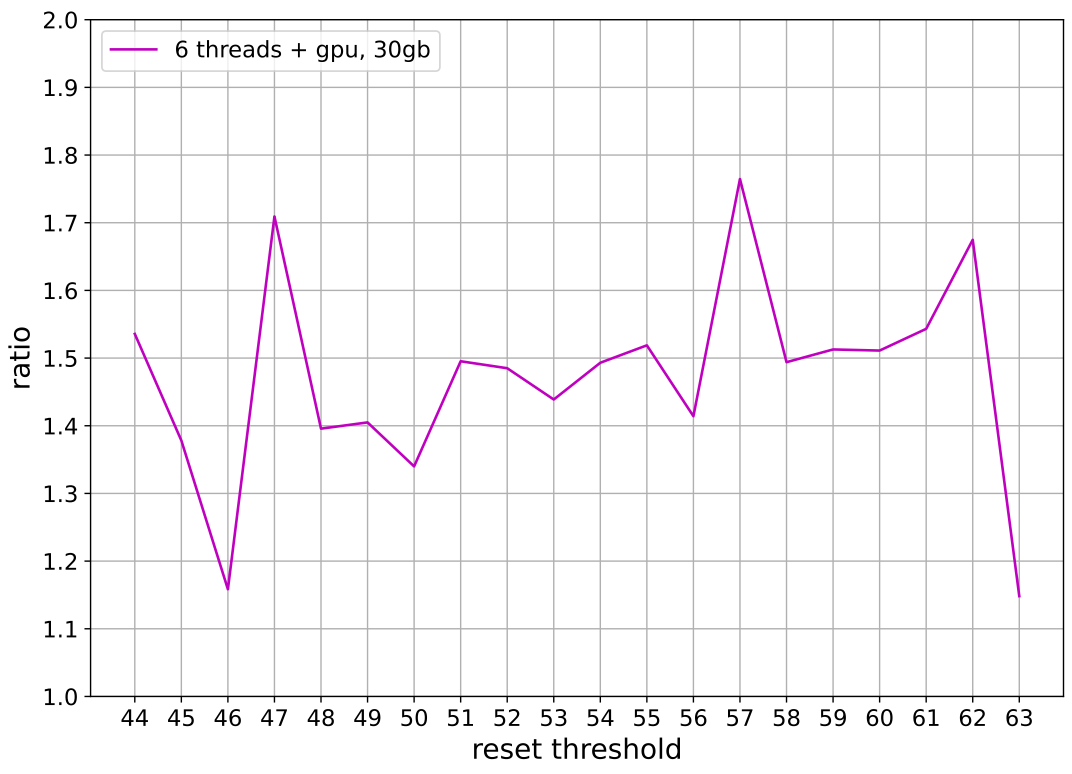

Figure 4.8: The ratio between average times for reset threshold $x + 1$ and x.

Chapter 5

Conclusions

The problem of computing the length of the shortest reset words is hard and the required time and space resources grow exponentially with the number of states in the automaton. We have improved the previously best known exact algorithm, which was unbeatable for several years, despite a few attempts and developed alternative solutions.

We obtained a substantial improvement by employing a series of algorithmic enhancements, which are also combined with technical low-level methods, thus working well in a practical setting. Decreasing the memory requirements was also crucial for processing larger automata, since after running out of memory, the algorithm has to switch to the DFS phase. If entered prematurely, this increases the running time very quickly, which happens much earlier in the old algorithm.

The experiments confirm that the new algorithm is significantly faster and deals with larger automata in a reasonable time, in contrast to the old algorithm. It is also noticeable that the required computation time grows slower in correspondence with the reset threshold, and this growth is stable and around 1.5 times for random binary automata.

Our computational package is an open tool that should be useful for researchers and engineers encountering the problem of synchronization of a DFA. It can be further generalized to deal with partial DFAs as well as with non-deterministic finite automata with various definitions of synchronizing words.

Our future work involves doing more experiments and probably further fine-tuning the algorithm, as well as some of the heuristics as the beam. For instance, with the new tool, we will be able to establish a better estimation of the expected reset threshold of a random automaton.

Bibliography

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

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

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

[4] 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.

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

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

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

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

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

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

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

[12] Daniel M. Kane and R. Ryan Williams. The orthogonal vectors conjecture for branching programs and formulas. *CoRR*, abs/1709.05294, 2017. URL: `http://arxiv.org/abs/1709.05294`, `arXiv:1709.05294`.

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

[14] 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.

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

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

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

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

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

[20] J. Olschewski and M. Ummels. The complexity of finding reset words in finite automata. In *Mathematical Foundations of Computer Science 2010*, volume 6281 of *LNCS*, pages 568–579. Springer, 2010.

[21] 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.

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

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

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

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

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

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

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

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

[30] 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.

[31] E. Skvortsov and E. Tipikin. Experimental study of the shortest reset word of random automata. In *Implementation and Application of Automata*, volume 6807 of *LNCS*, pages 290–298. Springer, 2011.

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

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

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

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

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

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

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

Chapter 6

Appendix: User guide

The following documentation can also be found in the `docs/` directory of the package.

6.1 Installation

To run the program you first need to install g++ (with -pthread support) and make. On Ubuntu 20.04 you can install all of these with:

```
$ sudo apt install g++ make libtbb-dev
```

If you want to enable GPU utilization, cuda and nvcc must also be installed. The following versions were used in testing (higher should be fine):

- g++ 9.3.0
- CUDA 11.4
- nvcc 10.1.243

To compile the program call make. Because the program uses just-in-time compilation, even though the first make succeeds, it might still fail to run in some or all configurations. The `-v/--verbose` option might be useful for checking what went wrong during runtime. Failed runs might cause compilation artefacts to be left in the `build/` folder. Delete them with make clean. To use the program from everywhere on your system, consider adding an alias `synchro='/path/to/repo/synchro'` to your `.bashrc`.

6.2 Usage

Call `synchro --help` to see the following message.

```
synchro [OPTION...]

  -f, --file arg    Path to the input file
  -c, --config arg  Path to the config file
  -o, --output arg  Path to the output file
  -v, --verbose     Verbose output
  -q, --quiet       Quiet output (only warnings and errors)
  -h, --help        Print usage
```

The input file must contain automata in the following format:

$$\begin{array}{l} N\ K \\ A_{0,0}\ A_{0,1}\ \dots\ A_{0,K-1}\ A_{1,0}\ A_{1,1}\ \dots \end{array}$$

where $A_{i,j}$ is the result of the transition function on i-th state and j-th letter. The states and letters indices are zero-based. The file can contain many inputs.

6.3 Example run

```
synchro --config configs/readme_config.json
        --file data/readme_input.txt -o save.txt
```

Output:

```
[22:54:39.650] [INFO] Read 4 automata
[22:54:39.651] [INFO] Recompiling for N = 1, K = 4
[22:54:44.261] [INFO] Minimum synchronizing word length: [0, 0]
[22:54:44.261] [INFO] Recompiling for N = 4, K = 3
[22:54:49.629] [WARNING@brute] Automaton is non-synchronizing
[22:54:49.629] [INFO] NON SYNCHRO
[22:54:49.629] [INFO] Loading precompiled library
[22:54:49.629] [INFO@brute] mlsw: 3
[22:54:49.629] [INFO] Minimum synchronizing word length: [3, 3]
[22:54:49.629] [INFO] Recompiling for N = 100, K = 3
[22:54:55.326] [INFO@brute] N > 20, exiting...
[22:54:55.326] [INFO@eppstein] Upper bound: 25
[22:54:55.326] [INFO@beam] Upper bound: 19
[22:54:55.337] [INFO@reduce] Reduced to N = 90 in 8 bfs steps
[22:54:55.339] [INFO] Recompiling for N = 90, K = 3
[22:55:00.987] [INFO@exact] mlsw: 18
[22:55:00.987] [INFO] Minimum synchronizing word length: [18, 18]
[22:55:00.987] [INFO] Saving synchronizing word of length 25
```

The output file contains information about synchronizing words in the following format:

```
[lower_bound, upper_bound] {w_0, w_1, ..., w_k}
```

In our example, the save.txt file should look like this

```
[0, 0]
NON SYNCHRO
[3, 3]
[18, 18] {0 0 1 0 1 0 1 0 2 2 1 0 1 2 2 0 0 2 2 0 0 2 0 1 2}
```

The default behavior of every algorithm is to exit if it cannot find a shorter synchronizing word than the predecessors. That is why in the first three cases, the Eppstein algorithm (which has the `find_word` parameter set to true) did not even run and the word was not saved. In the last case, the saved word has a length greater than 18, because Beam and Exact do not support the `find_word` parameter.

6.4 Configuration files

Json configuration files specify computational plans. A plan describes a sequence of algorithms with their parameters that are run sequentially. Parameters are either boolean, integer or string. Integer and boolean parameters can appear in json as a value of their respective type (e.g. `"find_word": true`) or as a string containing a valid C++ expression (e.g. `"find_word": "AUT_N < 1000 * 1000"`). The C++ expressions can use <`cmath`> functions and predefined `AUT_N`, `AUT_K` values, which respectively denote the number of states and the size of the alphabet of the given automaton. The only two exceptions to these rules are the `threads` and `gpu` global parameters, whose values **can not** be C++ expressions.

6.4.1 Global parameters

- `upper_bound` (integer) (default `"1ULL * AUT_N * AUT_N * AUT_N / 6"`) – Specifies the initial upper bound on reset threshold. Algorithms terminate if they do not find a reset word of length within this bound. Otherwise, the upper bound can be decreased by the algorithms.
- `threads` (integer) (default `1`) – Specifies the number of threads for parallel computation.
- `gpu` (boolean) (default `false`) – Enables gpu. Requies CUDA library and nvcc (see installation guide)

- `algorithms` (list) – Specifies the list of algorithms that the plan consists of. Algorithms will be run in the given order.

6.4.2 Algorithms

Each algorithm in the `algorithms` list is a dictionary with keys:

- `name` – One of the available algorithms (see the listbelow).
- `config` – Dictionary (possibly empty) containing algorithm parameters.

By default algorithms only improve upper and lower bounds on the shortest reset threshold. Some of them also support the `find_word` parameter, which makes them output the synchronizing word.

Brute

This is an exact algorithm for small automata. It runs BFS in the power automaton, using a bit array for visited sets. Thus, it requires O(2ˆN) memory.

- `max_n` (integer) (default `20`) – Specifies the maximum supported number of states. If the given automaton has more states, the algorithm does nothing. Must be less or equal to `32`.

Eppstein

The classic Eppstein algorithm works by running at most (n-1) iterations of a subprocedure, which chooses an arbitrary pair of not yet synchronized states and appends a word that synchronizes them to the result. The known upper bound on the length of the resulting reset word is cubic in the number of states.

- `transition_tables` (boolean) (default `false`) – Enables the precomputation of shortcuts in the pairs tree. Must be set to `false` if `find_word` is set to true.
- `find_word` (boolean) (default `false`) – Returns a reset word. If `false`, the algorithm only returns the length and sets the upper bound.

Beam

Implementation of the beam search algorithm. It runs inverse BFS in the power automaton. The initial list consists of all of the singletons. In each iteration, only `beam_size` best sets are kept (valued by their size).

- `beam_size` (integer) (default `"std::log2(AUT_N)"`)

- `max_iter` (integer) (default `-1`) – Limits the number of iterations (`-1` for unlimited, i.e., the `upper_bound` on reset threshold). Though very unlikely for random automata, beam search might run into a loop, especially for small `beam_size`. In this case, the algorithm would run for `upper_bound` number of iterations, which might be large if no previous algorithm decreased this bound.

- `presort` (string) (default `"none"`) – One of [`"none"`, `"indeg"`]. If `"indeg"` is specified, the algorithm permutes the indices of the automaton so that they are ordered from lowest to highest in-degree. Useful if the implementation uses tries.

Exact

Finds the reset threshold (the length of the shortest reset words) or computes a lower bound. The algorithm works by first running two BFS algorithms (one starting from the singletons, and one starting from the set of all the states) and checking in each iteration if they met. If the answer is not found in a certain amount of steps or the memory runs out, it optionally switches to a DFS algorithm. A good upper bound on the shortest reset threshold will decrease the running time greatly if the `dfs_shortcut` parameter is set to `true`.

- `dfs` (boolean) (default `true`) – Enables the DFS phase. If set to `false`, after the first phase fails to find the reset threshold, a lower bound is returned.

- `dfs_shortcut` (boolean) (default `true`) – Enables switching to the DFS phase when the algorithm calculates that it is better. If set to `false`, DFS is only entered after the memory runs out.

- `max_memory_mb` (integer) (default `2048`) – Maximum amount of used memory in megabytes. If `gpu` is set to true, this limit also applies to the GPU memory.

- `dfs_min_list_size` (integer) (default `10000`) – The minimum size of the list at each depth during the DFS phase.

- `strict_memory_limit` (boolean) (default `false`) – Stops the algorithm if there's not enough memory in the DFS phase. If set to `false`, only warnings are printed.

- `bfs_small_list_size` (integer) (default `"AUT_N * 16"`) – Until both BFS and I-BFS lists reach this size, the algorithm always picks the smaller side to expand and not consider DFS shortcut.

Reduce

Reduces the number of states of the automaton before entering the Exact algorithm (which is the only algorithm that can be run after Reduce succeeds). Works by running a small number of BFS iterations and then trying to delete all of the states that will never be reached again (outside the sink component).

- `min_n` (integer) (default `80`) – Specifies the minimum supported number of states. If the given automaton has fewer states, the reduction is skipped.
- `list_size_threshold` (integer) (default `"AUT_N * 16"`) – The algorithm tries to reduce the number of states after at least `list_size_threshold` sets are on the BFS list.