HAL open science

Synchronizing Heuristics: Speeding up the Slowest

Ömer Faruk Altun, Kamil Tolga Atam, Sertaç Karahoda, Kamer Kaya

To cite this version:

Ömer Faruk Altun, Kamil Tolga Atam, Sertaç Karahoda, Kamer Kaya. Synchronizing Heuristics: Speeding up the Slowest. 29th IFIP International Conference on Testing Software and Systems (ICTSS), Oct 2017, St. Petersburg, Russia. pp.243-256, 10.1007/978-3-319-67549-7_15 . hal-01678975

HAL Id: hal-01678975

https://inria.hal.science/hal-01678975v1

Submitted on 9 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Synchronizing Heuristics: Speeding Up The Slowest

Ömer Faruk Altun[1], Kamil Tolga Atam[1], Sertaç Karahoda[1], and Kamer Kaya[1,2]

[1] Computer Science and Engineering, Faculty of Engineering and Natural Sciences, Sabanci University, Tuzla, Istanbul, Turkey
{ofarukaltun,atam,skarahoda,kaya}@sabanciuniv.edu
[2] Dept. Biomedical Informatics, The Ohio State University, OH, USA

Abstract. Computing a shortest synchronizing word of an automaton is an NP–hard problem. Therefore, heuristics are used to compute short synchronizing words. SynchroP is among the best heuristics in the literature in terms of word lengths. The heuristic and its variants such as SynchroPL have been frequently used as a baseline to judge the quality of the words generated by the new heuristics. Although, its quality is good, the heuristics are significantly slow especially compared to much cheaper heuristics such as Greedy and Cycle. This makes them infeasible for large-scale automatons. In this paper, we show how one can improve the time performance of SynchroP and its variants by avoiding unnecessary computations which makes these heuristics more competitive than they already are. Our experimental results show that for 2500 states, SynchroP can be made 70–160× faster, via the proposed optimizations. In particular, for 2500 states and 32 letters, the SynchroP execution reduces to 66 seconds from 4745 seconds. Furthermore, the suggested optimizations become more effective as the number of states in the automata increase.
Keywords: Finite state automata, Synchronizing words, Synchronizing heuristics

1 Introduction

A *synchronizing word* w for an automaton A is a sequence of inputs such that no matter at which state A currently is, if w is applied, A is brought to a particular state. Such words do not necessarily exist for every automaton. An automaton with a synchronizing word is called *synchronizing automaton*.

Synchronizing automata have practical applications in many areas. For example in model based testing [1] and in particular for finite state machine based testing [2], test sequences are designed to be applied at a particular state. Note that a finite state machine given as the specification can be viewed as an automaton by omitting the output symbols labeling the transitions of the finite state machine. The implementation under test can be brought to the desired state by using a synchronizing word. Similarly, synchronizing words are used the

generate test cases for synchronous circuits with no reset feature [3]. Even when a reset feature is available, there are cases where reset operations are too costly to be applied. In these cases, a synchronizing word can be used as a compound reset operation [4]. Natarajan puts forward another surprising application area, part orienters, where a part moving on conveyor belt is oriented into a particular orientation by the obstacles placed along the conveyor belt [5]. The part is in some unknown orientation initially, and the obstacles should be placed in such a way that, regardless of the initial orientation of the part, the sequence of pushes performed by the obstacles along the way makes sure that the part is in a unique orientation at the end. Volkov presents more examples for the applications of synchronizing words together with a survey of theoretical results related to synchronizing automata [6].

As noted above, not every automaton is synchronizing. As shown by [7], checking if an automaton with n states and p letters is synchronizing can be performed in time $O(pn^2)$. For a synchronizing automaton, finding a shortest synchronizing word (which is not necessarily unique) is of interest from a practical point of view for obvious reasons (e.g., shorter test sequences in testing applications, or less number of obstacles for parts orienters, etc.).

The problem of finding the length of a shortest synchronizing word for a synchronizing automaton has been a very interesting problem from a theoretical point of view as well. This problem is known to be NP-hard [7], and coNP-hard [8]. Another interesting aspect of the problem is the following. It is conjectured that for a synchronizing automaton with n states, the length of the shortest synchronizing sequence is at most $(n-1)^2$, which is known as the *Černý Conjecture* in the literature [9, 10]. Posed half a century ago, the conjecture is still open and claimed to be one of the longest standing open problem in automata theory. The best upper bound known for the length of a synchronizing word is $(n^3-n)/6$ as provided by [11].

Due to the hardness results given above for finding shortest synchronizing words, there exist heuristics in the literature, known as *synchronizing heuristics*, to compute short synchronizing words. Among such heuristics are Greedy [7], Cycle [12], SynchroP [13], SynchroPL [13], and FastSynchro [14]. In terms of complexity, these heuristics are ordered as follows: Greedy/Cycle with time complexity $O(n^3+pn^2)$, FastSynchro with time complexity $O(pn^4)$, and finally SynchroP/SynchroPL with time complexity $O(n^5+pn^2)$ [13, 14], where n is the number of states and p is the size of the alphabet. This ordering with respect to the worst case time complexity is the same if the actual performance of the algorithms are considered (see for example [14, 15] for experimental comparison of the performance of these algorithms).

The SynchroP heuristic and its variants such as SynchroPL have been commonly used as a baseline to evaluate the performance of new heuristics in terms of synchronizing word length. However, since these heuristics are slow, a limited experimental setting with small-scale automata is usually employed for comparison purposes. For this reason, there exist attempts to improve the performance; for instance, a faster variant FastSynchro of SynchroP has been

proposed in the literature. FastSynchro proposes a cheaper way to choose path to follow while generating the synchronizing words. However, the performance improvement comes with an increase on the average length of the synchronizing words [13, 14].

In this work, we propose a set of techniques to make SynchroP much faster without changing its nature. Hence, the synchronizing words generated by the heuristic will be the same. The impact of the proposed techniques is two-fold: first, the SynchroP heuristic becomes more competitive to be used as a stronger benchmark for the new heuristics; our experimental results show that for 2500 states, SynchroP can be made 70–160× faster with our optimizations. Second, the heuristic becomes feasible to be used in practice; for instance, with 2500 states and 32 letters in the automaton, the execution time of the heuristic reduces to 66 seconds from 4745 seconds. Furthermore, the experiments reveal that suggested optimizations become more effective as the size of the automaton increases. As we will discuss later, it is straightforward to apply some of the proposed techniques to SynchroPL.

The rest of the paper is organized as follows: In Section 2, we introduce the notation used in the paper and explain SynchroP in detail. The proposed optimizations are introduced at Section 3 and experimental results are given in Section 4. Section 5 discusses threats to validity and Section 6 concludes the paper.

2 Background and Notation

A (complete and deterministic) *automaton* is defined by a triple $A = (S, \Sigma, \delta)$ where $S = \{1, 2, \ldots, n\}$ is a finite set of n states, Σ is a finite alphabet consisting of p input letters (or simply *letters*). $\delta : S \times \Sigma \to S$ is a transition function.

An element of the set $\Sigma^\star$ is called a *word*. For a word $w \in \Sigma^\star$, we use $|w|$ to denote the length of w, and ε is the empty word. We extend the transition function δ to a set of states and to a word in the usual way. We have $\delta(i, \varepsilon) = i$, and for a word $w \in \Sigma^\star$ and a letter $x \in \Sigma$, we have $\delta(i, xw) = \delta(\delta(i, x), w)$. For a set of states $C \subseteq S$, we have $\delta(C, w) = \{\delta(i, w) | i \in C\}$.

For a set of states $C \subseteq S$, let $C^2 = \{\langle i, j\rangle | i, j \in C\}$ be the set of all *multisets* with cardinality 2 with elements from C, i.e. C^2 is the set of all subsets of C with cardinality 2, where repetition is allowed. An element $\langle i, j\rangle \in C^2$ is called a *pair*. Furthermore, it is called *a singleton pair* (or *an s–pair*) if $i = j$, otherwise it is called *a different pair* (or *a d–pair*). The set of s–pairs and d–pairs in C^2 are denoted by C^2_s and C^2_d respectively.

A word w is said to be a *merging word for a pair* $\langle i, j\rangle \in S^2$ if $\delta(\{i, j\}, w)$ is singleton. Note that, for an s-pair $\langle i, i\rangle$, every word (including ε) is a merging word. A word w is called a *synchronizing word for an automaton* $A = (S, \Sigma, \delta)$ if $\delta(S, w)$ is singleton. An automaton A is called *synchronizing* if there exists a synchronizing word for A. In this paper, we only consider synchronizing automata. As shown by [7], deciding if an automaton is synchronizing can be

performed in time $O(pn^2)$ by checking if there exists a merging word for $\langle i,j\rangle$, for all $\langle i,j\rangle \in S^2$.

We use the notation $\delta^{-1}(i,x)$ to denote the set of those states with a transition to state i with letter x. Formally, $\delta^{-1}(i,x) = \{j \in S | \delta(j,x) = i\}$. We also define $\delta^{-1}(\langle i,j\rangle, x) = \{\langle k,\ell\rangle \mid k \in \delta^{-1}(i,x) \wedge \ell \in \delta^{-1}(j,x)\}$.

2.1 The SynchroP heuristic

SynchroP is composed of two phases. In the first phase, which is common to almost all existing heuristics, a shortest merging word $\tau_{\langle i,j\rangle}$ for each $\langle i,j\rangle \in S^2$ is computed by using a breadth first search such as the one given in Algorithm 1.

Algorithm 1: Computing shortest merging words for state pairs (Phase 1)

```
input  : An automaton A = (S, Σ, δ)
output: A shortest merging word τ_⟨i,j⟩ for all ⟨i,j⟩ ∈ S^2
let Q be an initially empty queue; // Q: BFS frontier
P = ∅; // P: the set of nodes in the BFS forest constructed so far
foreach ⟨i,i⟩ ∈ S_s^2 do push ⟨i,i⟩ onto Q, insert ⟨i,i⟩ into P, and set τ_⟨i,i⟩ = ε;
while P ≠ S^2 do // we still have some more pairs to discover
    ⟨i,j⟩ = pop the next item from Q;
    foreach x ∈ Σ do
        foreach ⟨k,ℓ⟩ ∈ δ^-1(⟨i,j⟩, x) do
            if ⟨k,ℓ⟩ ∉ P then
                τ_⟨k,ℓ⟩ = x τ_⟨i,j⟩;
                push ⟨k,ℓ⟩ onto Q;
                P = P ∪ {⟨k,ℓ⟩};
```

Algorithm 1 performs a breadth first search (BFS), and therefore constructs a BFS forest, rooted at s–pairs $\langle i,i\rangle \in S_s^2$, where these s–pair nodes are the nodes at level 0 of the BFS forest. A d–pair $\langle i,j\rangle$ appears at level k of the BFS forest if $|\tau_{\langle i,j\rangle}| = k$.

In almost all synchronizing heuristics, a second phase generates a synchronizing word in a constructive, step-by-step fashion. The heuristics keep track of the current set C of states, which is initially the entire set of states S. At each iteration, the cardinality of C is reduced at least by one. This is accomplished by picking a d-pair $\langle i,j\rangle \in C_d^2$, and considering $\delta(C, \tau_{\langle i,j\rangle})$ as the next active set in the next iteration. Since $\tau_{\langle i,j\rangle}$ is a merging sequence for (at least) the states i and j, the cardinality of $\delta(C, \tau_{\langle i,j\rangle})$ is guaranteed to be smaller than that of C. The synchronizing heuristics differ from each other in the way they pick the d-pair $\langle i,j\rangle \in C_d^2$ to be used at each iteration.

For a set of states $C \subseteq S$, let the cost $\phi(C)$ of C be defined as

$$\phi(C) = \sum_{i,j \in C} |\tau_{\langle i,j\rangle}|$$

$\phi(C)$ is a heuristic indication of how hard it is to bring the set C to a singleton. The intuition here is that, the larger the cost $\phi(C)$ is, the longer a synchronizing word would be required to bring C to a singleton set.

During the iterations of SynchroP, the selection of $\langle i,j\rangle \in C_d^2$ that will be used is performed by favoring the pair with the minimum possible cost $\delta(C, \tau_{\langle i,j\rangle})$. Based on this cost function, the second phase of SynchroP is given in Algorithm 2.

Algorithm 2: Computing a synchronizing word (Phase 2 of SynchroP)

input : An automaton $A = (S, \Sigma, \delta)$ and $\tau_{\langle i,j\rangle}$ for all $\langle i,j\rangle \in S^2$
output: A synchronizing word Γ for A

```
1  C = S; // C: current state set
2  Γ = ε; // Γ: synchronizing word to be constructed, initially empty
3  while |C| > 1 do // still not a singleton
4      minCost = ∞
5      foreach d–pair ⟨i,j⟩ ∈ C_d^2 do
6          thisPairCost = φ(δ(C, τ_⟨i,j⟩))
7          if thisPairCost < minCost then
8              minCost = thisPairCost
9              τ' = τ_⟨i,j⟩
10     Γ = Γ τ'; // append τ' to the synchronizing word
11     C = δ(C, τ'); // update current state set with τ'
```

3 Speeding up SynchroP and its Variants

In this section, we will introduce three improvements for increasing the performance of SynchroP. The first improvement explained in Section 3.1 precomputes the cost of $\delta(C, \tau_{\langle i,j\rangle})$ under certain conditions to eliminate some redundant cost computations. The improvement explained in Section 3.2 is in fact an improvement over the approach given in Section 3.1 where the precomputations are delayed until they are necessary. Finally in Section 3.3, we explain a particular improvement that can accelerate the first iteration of SynchroP, which in practice is the most expensive iteration of SynchroP.

3.1 Eliminating redundant cost computations

The first improvement is based on the following observation. For each d–pair $\langle i,j\rangle \in C_d^2$, the cost $\phi(\delta(C, \tau_{\langle i,j\rangle}))$ is calculated at line 6 of Algorithm 2. Suppose that for two different d–pairs $\langle i,j\rangle, \langle i',j'\rangle \in C_d^2$, we have $\tau_{\langle i,j\rangle} = \tau_{\langle i',j'\rangle}$. In this case, we surely have $\delta(C, \tau_{\langle i,j\rangle}) = \delta(C, \tau_{\langle i',j'\rangle})$. Therefore, computing the cost $\phi(\delta(C, \tau_{\langle i,j\rangle}))$ and $\phi(\delta(C, \tau_{\langle i',j'\rangle}))$ separately is a redundant work.

One approach to eliminate these redundant cost computations can be the following. For an integer $k \geq 1$, consider the set of non–empty words $\Sigma^{\leq k}$ of length at most k. Formally, $\Sigma^{\leq k} = \{\sigma \mid \sigma \in \Sigma^\star, 1 \leq |\sigma| \leq k\}$. In each iteration of SynchroP, one can precompute the cost $\phi(\delta(C, \sigma))$ for all $\sigma \in \Sigma^{\leq k}$. For any d–pair $\langle i,j\rangle \in C_d^2$, one can then simply look up the precomputed cost $\phi(\delta(C, \tau_{\langle i,j\rangle}))$ when $|\tau_{\langle i,j\rangle}| \leq k$. For a word $\sigma \in \Sigma^{\leq k}$, let $\Phi(\sigma)$ be this precomputed cost of $\phi(\delta(C, \sigma))$ for the current iteration with the active state set C. Although, the

values of $\phi(\delta(C,\sigma))$ and $\Phi(\sigma)$ are the same, the main difference is that ϕ is an expensive function and Φ is a data structure that stores a set of precomputed values of ϕ. Using the precomputed cost $\Phi(\sigma)$ for all $\sigma \in \Sigma^{\leq k}$, the second phase of SynchroP can be modified as shown in Algorithm 3.

Algorithm 3: Computing a synchronizing word (modified Phase 2 of SynchroP)

input : An automaton $A = (S, \Sigma, \delta)$ and $\tau_{\langle i,j \rangle}$ for all $\langle i,j \rangle \in S^2$, an integer $k \geq 1$
output: A synchronizing word Γ for A

```
C = S; // C: current state set
Γ = ε; // Γ: synchronizing word to be constructed, initially empty
while |C| > 1 do // still not a singleton
    foreach σ ∈ Σ^{≤k} do Φ(σ) = φ(δ(C, σ)); // precompute Φ(σ)
    minCost = ∞
    foreach d–pair ⟨i, j⟩ ∈ C_d^2 do
        if |τ_⟨i,j⟩| ≤ k then
            thisPairCost = Φ(τ_⟨i,j⟩)
        else
            thisPairCost = φ(δ(C, τ_⟨i,j⟩))
        if thisPairCost < minCost then
            minCost = thisPairCost
            τ' = τ_⟨i,j⟩
    Γ = Γ τ'; // append τ' to the synchronizing word
    C = δ(C, τ'); // update current state set with τ'
```

Although the improvement is always useful for eliminating duplicate computations in theory, one needs to be careful in practice. Indeed, the larger the value of k is, the more benefit one can obtain by eliminating such computations. However, the number of precomputed costs, and hence, the amount of memory to store the results of these computations also increase exponentially with k. Formally, for a given k, the number of different sequences whose costs are precomputed is equal to

$$K = \sum_{\ell=1}^{k} p^{\ell} = \frac{p^{k+1}-1}{p-1} - 1$$

where p is the alphabet size. We need to use $\Theta(K)$ space to store the precomputed costs. Let C be the active state set for the current iteration; each sequence τ can be applied with $\Theta(|C| \times |\tau|)$ automata accesses and the cost of the new state set $\delta(C,\tau)$ can be computed in $O(|C|^2)$ time and $O(|C|)$ extra memory to store the next active state set. Since there are K possible sequences in total, the overall cost of the precomputation phase for a single iteration is

$$O\left(|C| \sum_{\ell=1}^{k} \ell p^{\ell} + |C|^2 K\right) = O\left(|C| \frac{p-(k+1)p^{k+1}+kp^{k+2}}{(p-1)^2} + |C|^2 K\right).$$

To avoid the first part, we interleaved the automata accesses and cost computations; since $\Phi(\sigma)$ is computed for all $\sigma \in \Sigma^{\leq k}$, the state set $\delta(C, \sigma)$ can be stored and used to compute $\delta(C, \sigma x)$ with only $O(|C|)$ automata accesses for all $x \in \Sigma$ and $\sigma \in \Sigma^{<k}$. Overall, this yields $O(|C|K)$ automata accesses and $O(|C|^2 K)$ time complexity for a single iteration. This implementation requires $O(|C|k)$ extra space to store the intermediate active state sets.

3.2 Lazy computation of sequence costs

The approach explained in Section 3.1 precomputes $\Phi(\sigma)$ for all $\sigma \in \Sigma^{\leq k}$. However in an iteration of Algorithm 3, the only $\Phi(\sigma)$ values that we benefit from are the ones for which $\sigma = \tau_{\langle i,j \rangle}$ for some $\langle i, j \rangle \in C_d^2$. Therefore, rather than precomputing $\Phi(\sigma)$ for all $\sigma \in \Sigma^{\leq k}$, it is better if we could precompute $\Phi(\sigma)$ for only those $\sigma \in \Sigma^{\leq k}$ such that $\sigma = \tau_{\langle i,j \rangle}$ for some $\langle i, j \rangle \in C_d^2$.

One way of accomplishing this is to use a lazy computation approach to construct the data structure Φ. More explicitly, one can compute $\Phi(\sigma)$ for a $\sigma = \tau_{\langle i,j \rangle}$ the first time it is used in the iteration, and then store it for further uses in the same iteration. Algorithm 4 given below implements this approach.

Algorithm 4: Computing a synchronizing word (modified Phase 2 of SynchroP with lazy $\Phi(\sigma)$ computation

input : An automaton $A = (S, \Sigma, \delta)$ and $\tau_{\langle i,j \rangle}$ for all $\langle i, j \rangle \in S^2$, an integer $k \geq 1$
output: A synchronizing word Γ for A

```
C = S; // C: current state set
Γ = ε; // Γ: synchronizing word to be constructed, initially empty
while |C| > 1 do // still not a singleton
    foreach σ ∈ Σ^≤k do Φ(σ) = ∞;
    minCost = ∞;
    foreach d–pair ⟨i, j⟩ ∈ C_d^2 do
        if |τ_⟨i,j⟩| ≤ k then
            if Φ(τ_⟨i,j⟩) = ∞ then
                Φ(τ_⟨i,j⟩) = φ(δ(C, τ_⟨i,j⟩))
            thisPairCost = Φ(τ_⟨i,j⟩)
        else
            thisPairCost = φ(δ(C, τ_⟨i,j⟩))
        if thisPairCost < minCost then
            minCost = thisPairCost
            τ' = τ_⟨i,j⟩
    Γ = Γ τ'; // append τ' to the synchronizing word
    C = δ(C, τ'); // update current state set with τ'
```

Similar to the improvement described above, the space complexity for this improvement is also $\Theta(K)$ when a simple vector/array is used for Φ and the sequences are indexed and queried based on their ordered letters. Let C be the active state set in the current iteration. With lazy computation, the number of

different sequences, and hence, the number of cost computations, is bounded by the number of state pairs $\langle i, j\rangle \in C_d^2$. Considering $C = O(n)$, this yields a space complexity of $O(\min(K, n^2))$. This complexity can be easily obtained with a set or better with a hash table. Obviously, using such data structures will increase the query costs to the precomputed values. In our implementation, we use a simple vector for Φ that implies a $\Theta(K)$ complexity. However, we also select k in a way that makes $K = O(n^2)$ as described below.

Lazy computation does not have an impact on theoretical time complexity since all the cost computations are already meant to be done by the original SynchroP. That is there is no redundant cost computation incurred by the improvement. However, the k value still needs to be set to have a better memory utilization. To restrict the memory usage in a judicious way, we use the largest integer that satisfies

$$\left|\{\langle i, j\rangle \in S_d^2 : \tau_{\langle i,j\rangle} \in \Sigma^{\leq k}\}\right| \geq \sum_{\ell=1}^{k} p^{\ell}.$$

The right-hand of the inequality is the amount of memory that will be used and the left-hand side is the number of pairs in S_d^2 that can benefit from the improvement with maximum sequence length k. Since the left-hand side is $O(n^2)$, the memory complexity follows.

3.3 Accelerating the first iteration

The final improvement that will be suggested in this paper is based on the following observation.

Lemma 1. *Let $C \subseteq S$ be a subset of states and $\langle i, j\rangle, \langle i', j'\rangle \in C_d^2$ be two d–pairs such that $\tau_{\langle i,j\rangle} = \sigma\tau_{\langle i',j'\rangle}$ for some $\sigma \in \Sigma^\star$. If $\delta(C, \sigma) \subseteq C$ then $\phi(\delta(C, \tau_{\langle i,j\rangle})) \leq \phi(\delta(C, \tau_{\langle i',j'\rangle}))$.*

Proof. We have $\delta(C, \tau_{\langle i,j\rangle}) = \delta(\delta(C, \sigma), \tau_{\langle i',j'\rangle}) \subseteq \delta(C, \tau_{\langle i',j'\rangle})$, where the last step is due to the fact that $\delta(C, \sigma) \subseteq C$. Since $\delta(C, \tau_{\langle i,j\rangle}) \subseteq \delta(C, \tau_{\langle i',j'\rangle})$, we have $\phi(\delta(C, \tau_{\langle i,j\rangle})) \leq \phi(\delta(C, \tau_{\langle i',j'\rangle}))$.

Lemma 1 suggests that in an iteration of SynchroP if we have a set C, d–pairs $\langle i, j\rangle, \langle i', j'\rangle \in C_d^2$ satisfying the preconditions stated in Lemma 1, then we can eliminate the consideration of the d–pair $\langle i', j'\rangle$ in that iteration, since we will always have $\phi(\delta(C, \tau_{\langle i,j\rangle})) \leq \phi(\delta(C, \tau_{\langle i',j'\rangle}))$. Although it may feel highly unlikely to fulfill the preconditions of Lemma 1, Corollary 1 given below explains how Lemma 1 can easily be used in the first iteration of SynchroP.

Corollary 1. *For two d–pairs $\langle i, j\rangle, \langle i', j'\rangle \in S_d^2$ if $\tau_{\langle i,j\rangle} = \sigma\tau_{\langle i',j'\rangle}$ for some $\sigma \in \Sigma^\star$, then $\phi(\delta(S, \tau_{\langle i,j\rangle})) \leq \phi(\delta(S, \tau_{\langle i',j'\rangle}))$.*

Proof. Consider Lemma 1 when $C = S$.

Corollary 1 gives us the following improvement opportunity. In the first iteration of SynchroP, it is sufficient to consider only those d–pairs $\langle i, j\rangle \in S_d^2$ such that $\tau_{\langle i,j\rangle}$ is not a suffix of $\tau_{\langle i',j'\rangle}$ for any other d–pair $\langle i', j'\rangle \in S_d^2$. Notice how Algorithm 1 constructs the shortest merging sequences by using other shortest merging sequences as suffix at line 9.

3.4 Speeding up SynchroPL

The proposed techniques can be exploited also for SynchroP variants such as SynchroPL and FastSynchro. Let $C \subseteq S$ be the current active state set. For a sequence $\sigma \in \Sigma^*$, SynchroPL uses the cost function

$$\phi_{PL}(\delta(C,\sigma)) = \phi(\delta(C,\sigma)) + f(\sigma) = \sum_{i,j\in C} |\tau_{\langle i,j\rangle}| + f(\sigma)$$

where $f(.)$ is a function used to make the shorter sequences more preferable. It is suggested to use $f(\sigma) = |\sigma|$ where $|\sigma|$ denotes the length of the sequence σ [13]. The improvements based on precomputation and lazy computation can be easily adapted for this cost function. However, applying the last improvement is not straightforward since we omit the suffix sequences which are shorter than the sequences the improvement takes into account.

Using the proposed techniques with other cost functions such as the cardinality of active state sets, i.e., $\phi'(\delta(C,\sigma)) = |\delta(C,\sigma)|$, is also possible. However, the speedups for cheaper heuristics may not be as much as the ones that we obtain for SynchroP which we will show in the next section.

4 Experimental Results

All the experiments in the paper are performed on a single machine running on 64 bit CentOS 6.5 equipped with 64GB RAM and a dual-socket Intel Xeon E5-2620 v4 clocked at 2.10 GHz where each socket has 8 cores (16 in total) and 20MB cache. We only used a single core and all the speedups are obtained with no parallelization. The codes are compiled with `gcc 4.9.2` with the `-O3` optimization flag enabled.

To measure the impact of the proposed techniques, we used randomly generated automatons[3] with $n \in \{500, 1000, 1500, 2000, 2500\}$ states and $p \in \{2, 8, 32\}$ inputs. For each (n, p) pair, we randomly generated 5 different automata and executed each algorithm on them. The values in the figures and the tables are the averages of these 5 executions for each configuration.

4.1 Selecting the target to optimize

As described above, SynchroP has two phases where the first is common to many other synchronizing heuristics. In a previous study, we proposed algorithms

[3] For each state s and input x, $\delta(s, x)$ is randomly assigned to a state $s' \in S$.

to parallellize the first phase on a shared-memory multicore system [16]. The second phase is the one which makes SynchroP recognized as one of the slowest heuristics in the literature. This is why we, in this study, targeted this phase. We measured the execution times of the phases individually to observe the impact of the second phase's execution time to the overall execution time. As Table 1 shows, the second phase is responsible for almost all the execution time of the heuristic.

		n: number of states				
		500	1000	1500	2000	2500
	2	0.991	0.997	0.999	0.999	0.999
p	8	0.991	0.998	0.999	0.999	1.000
	32	0.982	0.995	0.998	0.999	0.999

Table 1: The ratio of the execution time of Phase 2 (Algorithm 2) to the overall execution time of SynchroP, i.e., Phase 1 (Algorithm 1) + Phase 2.

4.2 Impact of the proposed techniques

To measure the impact of the proposed techniques, we run them on the random automata we generated as explained above. Table 2 shows the results of these experiments. The timings in the table are for the whole heuristic, Phase 1 and Phase 2, for each variant. As the results show, the proposed improvements, especially lazy cost computation, reduce the runtime of SynchroP significantly and more than 100 speedups are obtained for some automata type. For each n and p, the exact speedups for each variant are given in Figure 1. As the trend of each subfigure shows, the impact of the proposed techniques increase with n. Although, the speedups seem to decrease with increasing p, the absolute difference between the naive SynchroP's execution time and those of the proposed variants increase.

As expected, each of the proposed techniques increases the performance, but with different amounts; the lazy cost computation is proven to be the most useful one. We later target the first iteration and added the third one described in Section 3.3 on top of lazy computation. Although its impact is not significant in practice, we were expecting more. Because, when the execution times of the Phase 2 iterations for the proposed lazy computation variant are measured, as Figure 2 shows, the first one dominates the overall execution time. Here the figure shows only the case for $n = 2500$. However, the same trend can be obtained for other automata sizes. We show the trend here for completeness and point out the bottleneck of our implementation for future studies. To overcome this bottleneck, other suffix or subset-based improvements can be applied. A promising one is representing an active state set with an unknown cost as a

		n: number of states				
	Algorithm	500	1000	1500	2000	2500
$p = 2$	Baseline [15]	6.2	72.0	324.5	969.1	2309.3
	Naive	2.6	30.4	133.3	382.5	881.7
	Precompute	1.3	10.0	67.5	108.6	308.7
	Lazy	0.2	0.9	2.1	4.3	7.7
	First Iter.	0.1	0.6	1.6	3.2	5.4
$p = 8$	Baseline [15]	9.5	123.1	682.8	2179.7	5440.8
	Naive	6.3	90.4	418.5	1247.3	2946.0
	Precompute	1.8	42.4	93.1	164.4	1687.2
	Lazy	0.3	1.7	8.6	19.9	33.1
	First Iter.	0.3	1.6	7.9	18.6	31.2
$p = 32$	Baseline [15]	12.9	162.5	785.3	2438.3	6085.4
	Naive	9.7	140.4	658.2	2008.7	4745.6
	Precompute	3.0	11.8	625.0	1113.9	1691.9
	Lazy	0.9	8.4	22.4	43.4	68.3
	First Iter.	0.9	8.2	22.0	42.1	66.7

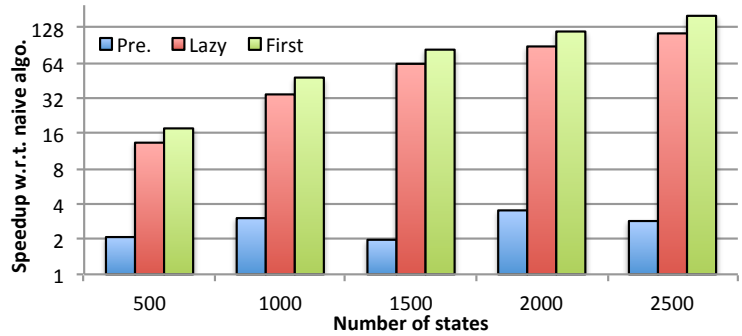
Table 2: The execution times of the SynchroP variants (in seconds) for $n \in \{500, 1000, 1500, 2000, 2500\}$ and $p \in \{2, 8, 32\}$. The first row for each p value is the baseline implementation from [15] and the second one is our baseline implementation. The next two rows are the variants with precomputation and lazy cost computation, respectively. The fifth and the last row is the one with additional first iteration optimization on top of lazy computation. Each value is average of five executions.

union/difference of other active sets whose costs are precomputed. This representation, with an efficient implementation, can be a great tool to reduce the number of cost computations.

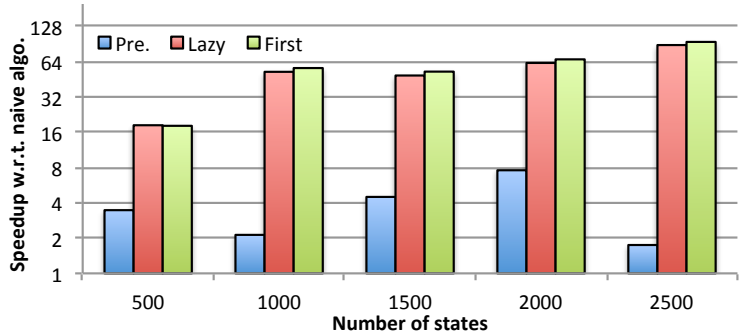
5 Threats to Validity

We consider several threats to validity of the methods suggested in this paper. First of all, to eliminate any implementation errors we may have in the new algorithms, we always check if a word w found by our implementations is a synchronizing word or not, by checking if $\delta(S, w)$ is singleton or not.

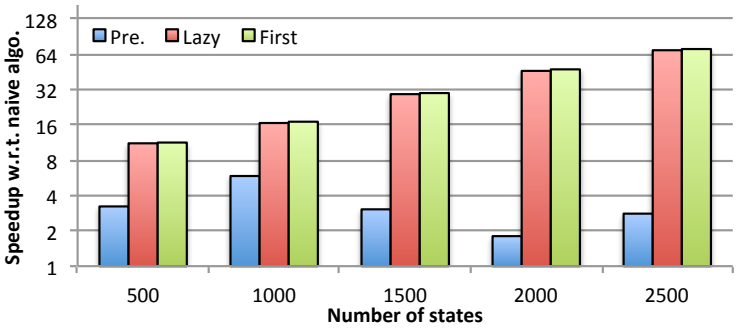
At each iteration, SynchroP selects a pair with minimum cost. Therefore the computed synchronizing sequence may change by picking a different pair with same cost. Algorithm 3 and 4 search the pair as in Algorithm 2, i.e. they pick the same pair by avoiding redundant computation. We also carefully implemented the variants in such a way that even the tie-breaking mechanisms become the same for all variants. In this way, we are able to check if the synchronizing words are the same for each variant which was the case in our experiments. On the other hand, the use of Corollary 1 can possibly eliminate some pairs with



(a) $p = 2$



(b) $p = 8$



(c) $p = 32$

Fig. 1: The speedup values normalized w.r.t. the naive SynchroP baseline for $n \in \{500, 1000, 1500, 2000, 2500\}$ and $p \in \{2, 8, 32\}$.

a minimum cost. Hence the algorithm may pick different pair with same cost. However we observed the same synchronizing sequences in our experiments.

Since we consider the speed ups over our naive SynchroP implementation, we need to be sure that our baseline implementation is competitive in terms of performance and word lengths. In this respect, we compared the synchronizing

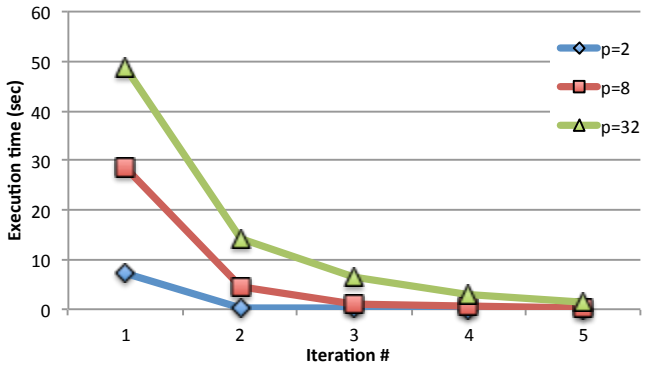


Fig. 2: The execution times of the iterations of the Lazy variant for $n = 2500$.

	n: # automata states				
p	500	1000	1500	2000	2500
2	78.6	111.2	147.4	160.6	192.8
8	45.4	70.2	85.6	98.6	111.2
32	37.8	54.8	66.6	78.2	88.0

Table 3: The length of the synchronizing sequences for $n \in \{500, 1000, 1500, 2000, 2500\}$ and $p \in \{2, 8, 32\}$.

word lengths of our naive implementation and those of [15] for 75 automata used in our experiments; the average ratio of the former to the latter is 1.01 for SynchroP, with a standard deviation of 0.02. In order to judge the time performance of our naive variant objectively, we also compared our naive implementation to the one in [15] as shown in Table 2. The comparison shows that our naive implementation is comparable to the state-of-the-art used in the literature.

6 Conclusion and Future Work

In this work, we proposed techniques to speedup SynchroP which is shown to produce shorter synchronizing words compared to cheaper heuristics such as Greedy and Cycle. Using various optimizations, we obtained order(s) of magnitude speed up for SynchroP. The techniques suggested in this paper become more effective as the size, i.e., the number of states, of the automata increases. With these improvements, SynchroP is more scalable and is highly practical even for automata with thousands of states.

Acknowledgments

This work was supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) [grant number 114E569].

We would like to thank the authors of [15] for providing their heuristics implementations, which we used to compare our naive baseline implementation as given in Table 2.

References

1. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.
2. David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
3. Hyunwoo Cho, Seh-Woong Jeong, F Somenzi, and C Pixley. Multiple observation time single reference test generation using synchronizing sequences. In *Design Automation, 1993, with the European Event in ASIC Design. Proceedings.[4th] European Conference on*, pages 494–498. IEEE, 1993.
4. Guy-Vincent Jourdan, Hasan Ural, and Hüsnü Yenigün. Reduced checking sequences using unreliable reset. *Inf. Process. Lett.*, 115(5):532–535, 2015.
5. B. K. Natarajan. An algorithmic approach to the automated design of parts orienters. In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 132–142. IEEE Computer Society, 1986.
6. Mikhail V Volkov. Synchronizing automata and the Černỳ conjecture. In *International Conference on Language and Automata Theory and Applications*, pages 11–27. Springer, 2008.
7. David Eppstein. Reset sequences for monotonic automata. *SIAM J. Comput.*, 19(3):500–510, 1990.
8. Jörg Olschewski and Michael Ummels. The complexity of finding reset words in finite automata. In *International Symposium on Mathematical Foundations of Computer Science*, pages 568–579. Springer, 2010.
9. Ján Černỳ. Poznámka k homogénnym experimentom s konečnỳmi automatmi. *Matematicko-fyzikálny časopis*, 14(3):208–216, 1964.
10. Ján Černỳ, Alica Pirická, and Blanka Rosenauerová. On directable automata. *Kybernetika*, 7(4):289–298, 1971.
11. Jean-Eric Pin. On two combinatorial problems arising from automata theory. *North-Holland Mathematics Studies*, 75:535–548, 1983.
12. A. N. Trahtman. Some results of implemented algorithms of synchronization. In *10th Journees Montoises d'Inform*, 2004.
13. Adam Roman. Synchronizing finite automata with short reset words. *Applied Mathematics and Computation*, 209(1):125–136, 2009.
14. R. Kudlacik, A. Roman, and H. Wagner. Effective synchronizing algorithms. *Expert Systems with Applications*, 39(14):11746–11757, 2012.
15. Adam Roman and Marek Szykula. Forward and backward synchronizing algorithms. *Expert Systems with Applications*, 42(24):9512–9527, 2015.
16. Sertaç Karahoda, Osman Tufan Erenay, Kamer Kaya, Uraz Cengiz Türker, and Hüsnü Yenigün. Parallelizing heuristics for generating synchronizing sequences. In Franz Wotawa, Mihai Nica, and Natalia Kushik, editors, *Testing Software and Systems - 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, volume 9976 of *Lecture Notes in Computer Science*, pages 106–122, 2016.