Synchronizing Heuristics For Weakly Connected Automata with Various Topologies

Berk Cirisci, Barış Sevilmiş, Emre Yasin Sivri, Poyraz Kıvanç Karaçam, Kamer Kaya and Hüsnü Yenigün

Computer Science and Engineering, Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey
{berkcirisci, barissevilmis, emreyasinsivri, karacam, kaya, yenigun}@sabanciuniv.edu

Keywords: Finite State Automata, Synchronizing Sequences, Strongly Connected Component.

Abstract: The problem of finding a synchronizing sequence for an automaton is an interesting problem studied widely in the literature. Finding a shortest synchronizing sequence is an NP-Hard problem. Therefore, there are heuristics to find short synchronizing sequences. Some heuristics work fast but produce long synchronizing sequences, whereas some heuristics work slow but produce relatively shorter synchronizing sequences. In this paper, we propose a method for using these heuristics by considering the strongly connectedness of automata. Applying the proposed approach of using these heuristics make the heuristics work faster than their original versions, without sacrificing the quality of the synchronizing sequences.

1 Introduction

A *synchronizing sequence w* for an automaton *A* is a sequence of inputs such that without knowing the current state of *A*, when *w* is applied to *A*, *A* reaches to a particular final state, regardless of its initial state. If an automaton *A* has a synchronizing sequence, *A* is called as *synchronizing automaton*.

Synchronizing automata and synchronizing sequences have various applications. One example area of application is the model-based testing, in particular Finite State Machine (FSM) based testing. When the abstract behavior of an interactive system is modeled by using an FSM, there are various methods to derive test sequences with high fault coverage [2, 4, 7]. These methods construct a test sequence to be applied when the implementation under test is at a certain state. Therefore, it is required to bring the implementation under test to this particular state, regardless of the initial state of the implementation, which can be accomplished by using a synchronizing

sequence. Even when the implementation has a reset input for this purpose, there are cases where using a synchronizing sequence is preferred [5]. For more examples of application areas of synchronizing sequences and for an overview of the theoretical results related to synchronizing sequences please see [11].

For practical purposes, e.g. the use of a synchronizing sequence in model-based testing, one is interested in finding synchronizing sequences as short as possible. However, finding a shortest synchronizing sequence is known to be a NP-hard problem [3]. Therefore, heuristic algorithms, known as synchronizing heuristics, are used to find short synchronizing sequences. Among such heuristics are Greedy [3], Cycle [10], SynchroP [8], and SynchroPL [6]. In this paper, we consider using the structure of an automaton while applying a synchronizing heuristic to speed up the execution of these heuristics. Namely, we consider the connectedness of automata.

An automaton is called *strongly connected* if every state is reachable from every other state by traversing the edges (only in the direction they point). An automaton is called *weakly connected* if one can reach any state starting from any other state by traversing the edges in some direction (i.e., not necessarily in the direction they point). To be synchronizing, an automaton needs to be at least weakly connected. When an automaton A is not strongly connected, it can be represented as a union of strongly connected automata. These automata are called as *strongly connected components* (*SCCs*) of A. In [1], given a weakly connected automaton A, a method is suggested to build a synchronizing sequence for A by using the synchronizing sequences of the SCCs of A. We will call the method suggested by [1] as *the SCC method*.

In [1], the SCC method has been experimented on randomly generated automata. Both Greedy and SynchroP algorithms have been applied to the strongly connected components of the automaton. The SCC method has proven itself by improving running time greatly in both algorithms. In [1], our aim was to find if taking the SCCs into account yields a better, faster heuristic, however we lacked variety in graph generation.

In this work, we focus on the performance of the SCC method further. We carefully construct our experimental testbed and connected the SCCs in various ways to generate different topologies. To have a control on the overall topology, we first assume a DAG of SCCs that can be either; 1) Linear, 2) All-To-One, 3) Tree, 4) Complete, 5) Random. Each vertex in the DAG corresponds to a single SCC of the automata. These DAG structures are described in more detail later in the text. While we connect the SCCs, we respect the chosen DAG and do not allow any other connections between SCCs. Our main interest is on the possible improvements on the sequence length and running time on the above-mentioned DAG types. To analyze the method's performance further, we also experiment on the ratio of the number of the edges connecting SCC's.

The rest of the paper is organized as follows. In Section 2, we introduce the notation and briefly give the required background. In Section 3, we introduce our approach. In Section 4, we talk about the synchronizing heuristics that we have worked on and their integration to our approach. In Section 5, we compare the proposed approach with the traditional one that performs synchronization heuristics on full automata. In Section 6, we conclude the paper and provide some future directions for our work.

2 Background and Notation

A (deterministic) *automaton* is defined by a tuple $A = (S, \Sigma, D, \delta)$ where S is a finite set of n states, Σ is a finite alphabet consisting of p input letters (or simply *letters*). $D \subseteq S \times \Sigma$ is the called the *domain* and $\delta: D \rightarrow S$ is a transition function. When $D = S \times \Sigma$, then A is called *complete*, otherwise A is called *partial*. Below, we consider only complete automata, unless otherwise stated.

An element of the set Σ^* is called an *input sequence* or simply *a sequence*. For a sequence $w \in \Sigma^*$, we use $|w|$ to denote the length of w, and ε is the empty sequence of length 0. For a complete automaton, we extend the transition function δ to a set of states and to a sequence in the usual way. For a state $s \in S$, we have $\delta(s, \varepsilon) = s$, and for a sequence $w \in \Sigma^*$ and a letter $x \in \Sigma$, we have $\delta(s, xw) = \delta(\delta(s, x), w)$. For a set of states $C \subseteq S$, we have $\delta(C, w) = \{\delta(s, w) \mid s \in C\}$.

For a set of states $C \subseteq S$, let $C^2 = \{\{s, s'\} \mid s, s' \in C\}$ 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 $\{s, s'\} \in C^2$ is called a *pair*. Furthermore, it is called *a singleton pair* (or *an s–pair,* or simply a *singleton*) if $s = s'$, 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 sequence w is said to be a *merging sequence for a pair* $\{s, s'\} \in S^2$ if $\delta(\{s, s'\}, w)$ is singleton. For an s-pair $\{s, s\}$, every sequence (including ε) is a merging sequence. For a given automaton $A = (S, \Sigma, S \times \Sigma, \delta)$ and a subset of states $S' \subseteq S$, a sequence w is called an *S'-synchronizing sequence for A* if $\delta(S', w)$ is singleton. When $S' = S$, w is simply called a *synchronizing sequence for A*. An automaton A is called *S'-synchronizing* if there exists an S'-synchronizing sequence for A. An automaton A is called *synchronizing* if there exists a synchronizing sequence for A.

In this paper, we only consider synchronizing automata. As shown by Eppstein [3], deciding if an automaton is synchronizing can be performed in time $O(pn^2)$ by checking if there exists a merging sequence for $\{s, s'\}$, for all $\{s, s'\} \in S^2$.

We use $\delta^{-1}(s, x)$ to denote the set of those states with a transition to state s with letter x. Formally, $\delta^{-1}(s, x) = \{s' \in S \mid \delta(s', x) = s\}$. For pairs, we also define $\delta^{-1}(\{s, s'\}, x) = \{\{p, p'\} \mid p \in \delta^{-1}(s, x) \wedge p' \in \delta^{-1}(s', x)\}$.

An automaton $A = (S, \Sigma, S \times \Sigma, \delta)$ is said to be *strongly connected* if for every pair of states $s, s' \in S$, there exists a sequence $w \in \Sigma^*$ such that $\delta(s, w) = s'$. Given an automaton $A = (S, \Sigma, S \times \Sigma, \delta)$ and another automaton $B = (S', \Sigma, D, \delta)$, B is said to be a *sub-automaton* of A if (i) $S' \subseteq S$, (ii) $D = \{(s, x) \in S' \times \Sigma \mid \exists s' \in S' \text{ s.t. } \delta(s, x) = s'\}$, and (iii) $\forall (s, x) \in D$, $\delta(s, x) = \delta(s, x)$. Intuitively, the states of B consist of a subset of states of A. Every transition in A from a B state to a B state is preserved, and all the other transitions are deleted.

A *strongly connected component* (*SCC*) of a given automaton $A = (S, \Sigma, S \times \Sigma, \delta)$, is a sub-automaton $B = (S, \Sigma, D, \delta)$ of A such that, B is strongly connected, and there does not exist another strongly connected sub-automaton C of A, where B is a sub-automaton of C. When one considers an automaton A as a directed graph (by representing the states of A as the nodes, and the transition between the states as the edges of the graph), B simply corresponds to a SCC of the directed graph of A.

For a set of SCCs $\{A_1, A_2, \ldots, A_k\}$, where $A_i = (S_i, \Sigma, D_i, \delta_i)$, $1 \leq i \leq k$, we have $S_i \cap S_j = \emptyset$ when $i \neq j$, and $S_1 \cup S_2 \cup \ldots \cup S_k = S$. Please note here that $k = 1$ if and only if A is strongly connected. In Figure 1, a weakly connected automaton and its SCCs are given.

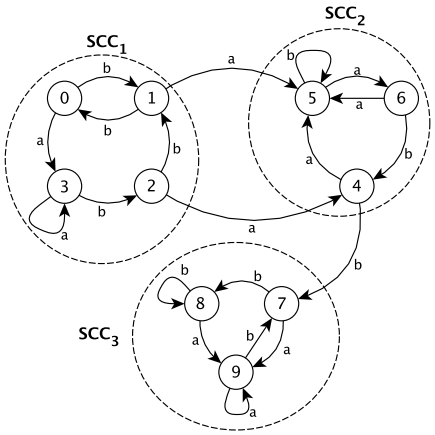

Figure 1: An automaton with 10 states, 2 inputs and 3 SCCs. (Source: [1])

An SCC $A_i = (S_i, \Sigma, D_i, \delta_i)$ is called a sink component if $D_i = S_i \times \Sigma$. In other words, for a sink component, all the transitions of the states in S_i in A are preserved in A_i. Therefore, if $A_i = (S_i, \Sigma, D_i, \delta_i)$ is not a sink component, then some transitions of some states will be missing. For this reason, A_i is a complete automaton if and only if A_i is a sink component. For the automaton given in Figure 1, SCC3 is the only sink component.

Given a partial automaton, we consider the completion of this automaton by introducing a new state and adding the missing transitions of states to this new state. Formally for a partial automaton $A = (S, \Sigma, D, \delta)$ such that $D \subset S \times \Sigma$, we define *the completion of A* as $A' = (S \cup \{*\}, \Sigma, S \times \Sigma, \delta')$, where (i) the star state * is a new state which does not exist in S, (ii) $\forall (s, x) \in D$, $\delta'(s, x) = \delta(s, x)$, (iii) $\forall (s, x) \notin D$, $\delta'(s, x) = *$, (iv) $\forall x \in \Sigma$, $\delta'(*, x) = *$. Any SCC that is not a sink component will be a partial automaton. In Figure 2, the completion of the SCC1 of the automaton of Figure 1 is given.

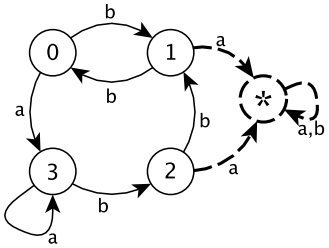

Figure 2: $SCC_1(A_1)$ with additional star state. * is the synchronizing state. (Source: [1])

3 Sequences for Weakly Connected Automata

To keep the paper self-contained, we explain the SCC method given in [1]. Consider an automaton $A = (S, \Sigma, S\times\Sigma, \delta)$ and its SCC decomposition $\{A_1, A_2, \ldots, A_k\}$.

Lemma 1: A is synchronizing iff there exists only one sink component in A_i in $\{A_1, A_2, \ldots, A_k\}$ and A_i is synchronizing.

Proof: If there are multiple sink components A_i and A_j, a state s_i and A_i and a state s_j in A_j cannot be merged, hence A is not synchronizing. If A_i is the only sink component of A and A_i is not synchronizing, A is not synchronizing as well.

Similar to the topological sorting of strongly connected components of a graph, we consider the topological sorting of SCCs of an automaton. Let $A = (S, \Sigma, S\times\Sigma, \delta)$ be an automaton and $\{A_1, A_2, \ldots, A_k\}$ be the SCCs of A. We consider the SCCs of A sorted as $\langle A_1, A_2, \ldots, A_k\rangle$ such that for any $1 \le i < j \le k$, there do not exist $s_i \in S_i$, $s_j \in S_j$, $w \in \Sigma^*$ where $\delta(s_j, w) = s_i$. Note that in this case A_k must be a sink component. From this point on, we consider $A = (S, \Sigma, S\times\Sigma, \delta)$ to be an automaton which is not strongly connected, and we consider $\langle A_1, A_2, \ldots, A_k\rangle$ to be the topologically sorted SCCs of A, where $A_i = (S_i, \Sigma, D_i, \delta_i)$. We have the following result.

Lemma 2: For any sequence $w \in \Sigma^*$ and for a state $s \in S_i$, $1 \le i \le k$, we have $\delta(s, w) \in (S_i \cup S_{i+1} \cup \ldots \cup S_k)$.

Proof: Since the components are topologically sorted, states in A_i can only move to a state in A_i, or to a state in $A_{i+1}, A_{i+2}, \ldots, A_k$.

Lemma 3: Let A_i be an SCC of an automaton which is not a sink component. Then the completion A'_i of A_i is a synchronizing automaton.

Proof: When A_i is not a sink component, then A_i is a partial automaton. In this case the completion A'_i of A_i, is an automaton with a unique sink-state (the star state), which is known to be synchronizing [9, 12].

3.1 An Initial Approach to Use SCCs

We now explain an idea to form a synchronizing sequence for an automaton A by using synchronizing sequences of the SCCs of A. For $1 \le i < k$, let β_i be a synchronizing sequence for the completion A'_i of $A_i = (S_i, \Sigma, D_i, \delta_i)$. Based on Lemma 3, one can always find a synchronizing sequence for A'_i, $1 \le i < k$. Let β_k be a synchronizing sequence for A_k. Lemma 1 suggests that A_k always has a synchronizing sequence if A is synchronizing.

We first claim that the sequence $\beta_1\beta_2\ldots\beta_k$ is a synchronizing sequence for A. In order to see this, it is sufficient to observe the following.

Lemma 4: For any $0 \le i < k$ we have $\delta(S, \beta_1\beta_2\ldots\underline{\beta_i}) \subseteq (S_{i+1} \cup S_{i+2} \cup \ldots \cup S_{\underline{k}})$

Proof: We will use induction; the base case, $i = 0$, holds trivially. Assume that the claim holds for i-1, i.e. $\delta(S, \beta_1\beta_2\ldots\beta_{i-1}) \subseteq (S_i \cup S_{i+1} \cup \ldots \cup S_{\underline{k}})$. For a state $s \in \delta(S, \beta_1\beta_2\ldots\beta_{i-1})$ such that $s \in (S_{i+1} \cup S_{i+2} \cup \ldots \cup S_{\underline{k}})$, then $\delta(s,\beta_i)$ will also belong to $(S_{i+1} \cup S_{i+2} \cup \ldots \cup S_{\underline{k}})$ based on Lemma 2. Therefore it remains to show that for any state $s \in \delta(S, \beta_1\beta_2\ldots\beta_{i-1})$ such that $s \in S_i$, $\delta(s, \beta_i)$ is not in S_i. The sequence β_i is a synchronizing sequence for the completion A'_i of SCC A_i. Since the star state of A'_i is the only state in which the states of A'_i can be synched, we must have $\delta'_i(S_i, \beta_i) = \{*\}$. Note that the star state in A'_i represents the states $S\backslash S_i$ for A_i. Hence the sequence β_i is in fact a sequence

that pushes all the states in S_i to the states in the other components, i.e., $\delta(S_i, \beta_i) = \emptyset$. This implies that for a state $s \in \delta(S, \beta_1\beta_2\ldots\beta_{i-1})$ such that $s \in S_i$, $\delta(s, \beta_i)$ is not in S_i. Finally, we can state the following result.

Theorem 5: Let β_i be a synchronizing sequence for the completion A'_i of A_i, $1 \leq i < k$, and let β_k be a synchronizing sequence for A_k. The sequence $\beta_1\beta_2\ldots\beta_k$ is a synchronizing sequence for A.

Proof: Using Lemma 4, we have $\delta(S, \beta_1\beta_2\ldots\beta_{k-1}) \subseteq S_k$. Since β_k is a synchronizing sequence for A_k, $\delta(S_k, \beta_k)$ is singleton. Combining these two results, we have $\delta(\delta(S, \beta_1\beta_2\ldots\beta_{k-1}), \beta_k) = \delta(S, \beta_1\beta_2\ldots\beta_{k-1}\beta_k)$ singleton as well.

3.2 An Improvement on the Initial Approach

Theorem 5 implies a trivial algorithm for constructing a synchronizing sequence for an automaton A based on its SCCs. As one may notice, though, the length of the sequence to be constructed can be reduced based on the following observation. Consider a sequence β_i, for some $1 < i \leq k$, used in the sequence $\beta_1\beta_2\ldots\beta_{k-1}\beta_k$. The sequence β_i is constructed to push all the states in A_i out of the component A_i. However, the sequence $\beta_1\beta_2\ldots\beta_{i-1}$ applied before β_i can already push some of the states in A_i out of A_i. On the other hand, the sequence $\beta_1\beta_2\ldots\beta_{i-1}$ can also move some of the states in the components $A_1, A_2, \ldots, A_{i-1}$ to a state in A_i. Therefore, a more careful approach can be taken considering which states in A_i must be moved out of A_i when constructing the sequence to handle the component A_i.

To take this observation into account, we define the following sequences recursively. For the base cases, we define $\alpha_0 = \varepsilon$ and $\sigma_0 = \varepsilon$. For $1 \leq i < k$, let $S'_i = S_i \cap \delta(S, \sigma_{i-1})$ and let α_i be a S'_i-synchronizing sequence for A'_i. For $1 \leq i < k$, let $\sigma_i = \sigma_{i-1}\alpha_i$.

Lemma 6: For $0 \leq i < k$, $\delta(S, \sigma_i) \subseteq S_{i+1} \cup S_{i+2} \cup \ldots \cup S_k$.

Proof: For the base case i=0 we have $\delta(S, \sigma_0) = \delta(S, \varepsilon) = S = S_1 \cup S_2 \cup \ldots \cup S_k$, so it trivially holds. $\delta(S, \sigma_i) = \delta(S, \sigma_{i-1}\alpha_i) = \delta(\delta(S, \sigma_{i-1}),\alpha_i)$ and using the induction hypothesis we have $\delta(S, \sigma_{i-1}) \subseteq S_i \cup S_{i+1} \cup \ldots \cup S_k$. α_i is a S'_i-synchronizing sequence for A'_i, where $S'_i = S_i \cap \delta(S, \sigma_{i-1})$. Therefore, α_i merges all the states in S'_i in the star state of A_i, which means $\delta(S'_i, \alpha_i) = \emptyset$. Therefore $\delta(\delta(S, \sigma_{i-1}),\alpha_i) \subseteq S_{i+1} \cup S_{i+2} \cup \ldots \cup S_k$.

Theorem 7: Let $S'_k = S_k \cap \delta(S, \sigma_{k-1})$ and α_k be a S'_k-synchronizing sequence for A_k. Then $\sigma_{k-1}\alpha_k$ is a synchronizing sequence for A.

Proof: If α_k is a S'_k-synchronizing sequence for A_k, $\delta(S'_k, \alpha_k)$ is a singleton. Using Lemma 6, we have $\delta(S, \sigma_{k-1}) \subseteq S_k$. Therefore $S'_k = S_k \cap \delta(S, \sigma_{k-1}) = \delta(S, \sigma_{k-1})$. Then we have $\delta(S, \sigma_{k-1}\alpha_k) = \delta(\delta(S,\sigma_{k-1}),\alpha_k) = \delta(S'_k,\alpha_k)$, which is a singleton.

Based on Theorem 7, the algorithm given in Figure 3 can be used to construct a synchronizing sequence for an automaton A. This algorithm uses a synchronizing heuristic to find a synchronizing sequence for SCCs. Any synchronizing heuristic can be used for this step. In the next section, we explain two different algorithms from the literature that we used in our experiments.

```
Input:   An automaton A = (S,Σ,D,δ)
Output: A synchronizing sequence for A

C = S; // All states are active initially
Γ = ε ;  // Γ: synch. sequence to be constructed,
         // initially empty
<A1, A2,…, Ak> = find/sort SCCs of A
foreach i in {1, 2, …, k} do
      // Consider Ai = (Si,Σ,Di,δi)
      S'i = C ∩ Si; // find active states of Ai
      Γi = Heuristic(A'i,S'i); // find S'i sync. sequence
                               // of completion A'i of Ai
      Γ = Γ Γi; // append Γi to sync. seq.
      C = δ(C, Γi ); // Update active states
return Γ ;
```

Figure 3: SCC algorithm to compute synchronizing sequences. (Source: [1])

4 Synchronizing Heuristics

There exist various synchronizing heuristics in the literature where we experimented with two of these heuristics, *Greedy* and *SynchroP*. Both of these heuristics have two phases. Phase 1 is common in these heuristics and given in Figure 4 below.

```
Input:   An automaton A = (S,Σ,D,δ)
Output: A merging sequence for all
         {i,j} ∈ S²

let Q be an initially empty queue // Q: BFS frontier
P = Ø // P: keeps the set of nodes in the BFS forest constructed so far
foreach {i,j} ∈ S²s do
   push {i,j} onto Q
   insert {i,j} into P
   set τ(i,j) = ε;

while P ≠ S² do
   {i,j}= pop 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}};
```

Figure 4: Phase 1 of Greedy and SynchroP. (Source: [1])

In Phase 1 of the synchronizing heuristics, a shortest merging sequence $\tau(i, j)$ for each $\{i, j\} \in S^2$ is computed by using a breadth first search. Note that in the algorithm, $\tau(i,j)$ is not unique.

Figure 4 performs a breadth first search (BFS), and therefore constructs a BFS forest, rooted at s–pairs $\{i, i\} \in S^2_s$, where these s–pair nodes are the nodes at level 0 of the BFS forest. A d–pair $\{i, j\}$ appears at level k of the BFS forest if $|\tau\{i,j\}| = k$. The first phase requires $\Omega(n^2)$ time since each $\{i, j\} \in S^2$ is pushed to Q exactly once.

4.1 The Greedy Heuristic

Greedy's Phase 2 (given in Figure 5 below) constructs a synchronizing sequence by using the information from Phase 1. Its main loop can iterate at most $n - 1$ times, since in each iteration $|C|$ is reduced by at least one. The min operation at line 4 requires $O(n^2)$ time and line 5 takes constant time. Line 6 can normally be handled in $O(n^3)$ time, but using the information precomputed by the intermediate stage between Phase 1 and Phase 2 [3], line 6 can be handled in O(n) time. Therefore, Phase 2 of Greedy requires $O(n^3)$ time. Note that Phase 2 of Greedy finds an S-synchronizing sequence for a given complete automaton $A = (S, \Sigma, S\times\Sigma, \delta)$. However, we need to find an S'-synchronizing sequence for a given subset $S' \subseteq S$ of states.

```
Input: An automaton A = (S,Σ,D,δ), τ(i,j) for all {i,j} ∈ S²s, S′ to be
synchronized
Output: An S′-synch. sequence Γ for A

C = S′   // C: current state set
Γ = ε    // Γ: synch. sequence to be constructed, initially empty

while |C| > 1 do  // still not a singleton
   {i,j} = arg min<k,l>∈C²d|τ(k,l)|; // decide the d-pair to be
                                     // merged
   Γ = Γ τ(i,j);  // append τ(i,j) to the synchronizing sequence
   C= δ(C, τ(i,j));     //update current state set with τ(i,j)
```

Figure 5: Phase 2 of Greedy. (Source: [1])

4.2 The SynchroP Heuristic

Similar to Greedy, the second phase of SynchroP constructs a synchronizing sequence iteratively. It keeps track of the current set C of states, which is initially the entire set of states S. In each iteration, the cardinality of C is reduced at least by one. This is accomplished by picking a d-pair $\{i, j\} \in C^2_d$ in each iteration, and considering $\delta(C, \tau(i, j))$ as the current set in the next iteration. Since $\tau(i, j)$ is a merging sequence for the states i and j, the cardinality of $\delta(C, \tau(i, j))$ is guaranteed to be smaller than that of C. For a set of states $C \subseteq S$, let the cost $\varphi(C)$ of C be defined as

$$\varphi(C) = \sum_{i,j \in C} |\tau(i,j)| \quad (1)$$

where $\varphi(C)$ 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 $\varphi(C)$ is, the longer a synchronizing sequence would be required to bring C to a singleton set.

During the iterations of SynchroP, the selection of $\{i, j\} \in C^2_d$ that will be used is performed by considering the cost of the set $\delta(C,\tau(i, j))$. Based on this cost function, the second phase of SynchroP is given in Figure 6. As in Greedy with the SCC method, we also use a slightly modified version of the second phase of SynchroP algorithm to find S'-synchronizing sequence.

```
Input:   An atomaton A = (S,Σ,D,δ), τ(i,j) for all {i,j} ∈ S²s, S′ to
be synchronized
Output: An S′-synch. sequence Γ for A

C = S′    // C: current state set
Γ = ε     // Γ: synchronizing sequence to be constructed,
          // initially empty
while |C| > 1 do  // still not a singleton
   minCost = ∞
   foreach d-pair {i,j} ∈ C²d do
      thisPairCost = Φ(δ(C,τ(i,j)))
      if thisPairCost < minCost then
         minCost = thisPairCost
         τ′ = τ(i,j)

   Γ = Γ τ′;   // append τ′ to the synch. sequence
   C= δ(C, τ′); //update current state set with τ′
```

Figure 6: Phase 2 of SynchroP. (Source: [1])

5 Experimental Results

The experiments were performed on a machine with Intel Xeon E5-2620 CPU and 64GB of memory, using Ubuntu 16.04.2. The code was written in C/C++ and compiled using gcc with -O3 option enabled.

In order to evaluate the performance of the SCC method, we generated automata with 5 different DAG types, $nSCC \in \{32, 64, 128\}$ states for each SCC, $p \in \{2, 4, 8\}$ inputs, $d \in \{0.25, 0.5, 0.75\}$ (edge distribution factor), $k \in \{2, 4, 8\}$ SCC's for four DAG types ($t \in$ {All-to-One, Linear, Complete, Random}) and $k \in \{3, 7, 15\}$ SCC's for one DAG type (t = Tree). To construct an automaton A with the given parameters, we first construct k different strongly connected automata $A_1, A_2, \ldots, A_k$, where each A_i has $nSCC$ states, p inputs. To construct an automaton A_i, we consider each state s in A_i and each input x, and assign $\delta(s, x)$ to be one of the states in A_i randomly. If A_i is not strongly connected after the initial random assignment, we reassign $\delta(s, x)$ for some of the states and inputs randomly again, keep repeating this process until A_i becomes strongly connected. Once we have A_i strongly connected, we identify those state s and input x pairs in A_i (except for the last SCC A_k) such that A_i stays strongly connected even without using the transition of the state s and with the input x. For these state/input pairs in A_i, we reassign $\delta(s, x)$ to be one of the states in the automata $A_{i+1}, A_{i+2}, \ldots, A_k$ according to the DAG type that we select. We used the distribution factor parameter $d \in \{0.25, 0.5, 0.75\}$ as follows: after finding M transitions that can be reassigned safely without making A_i not strongly connected, we use at most ($d \times M$) of them to connect the current SCC to the other SCCs based on the chosen DAG type t.

For Linear automata, we connect each SCC A_i to the SCC A_{i+1} except A_k as shown in Figure 7. For Complete automata, all SCC A_i to SCC A_j connections are established for $i < j$. For Random automata, each transition reassignment for SCC A_i is performed randomly to an SCC A_j where $i < j$. In All-to-one automata, all the reassignments are performed to *SCC* A_k, the sink component. For Tree automata, we connect each SCC

A_i to its parent in a (complete binary) tree. In this automata type, $k \in \{3, 7, 15\}$ since we aim to generate complete trees.

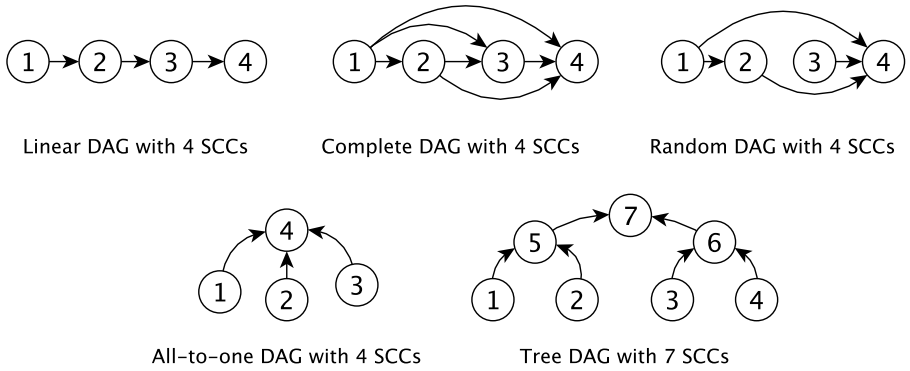

Figure 7: Example DAGs with 4 SCCs for the types Linear, Complete, Random and All-to-one, and 7 SCCs for the type Tree.

In general, for each *nSCC-p-d-k-t* combination we created 50 automata. However, due to the complexity of SynchroP, for automata with $nSCC = 128$, $p \in \{2, 4, 8\}$ inputs, $d \in \{0.25, 0.5, 0.75\}$, $k = 15$ and $t =$ Tree, we created only 20 automata rather than 50.

For an automaton $A = (S, \Sigma, S\times\Sigma, \delta)$ with *nSCC* states for each SCC, p inputs and k SCCs $\langle A_1, A_2, \ldots, A_k\rangle$ where $A_i = (S_i, \Sigma, D_i, \delta_i)$, $1 \le i \le k$, we find a synchronizing (i.e. S-synchronizing) sequence for A by using Greedy and SynchroP algorithms given in Figure 5 and Figure 6, respectively. We also find a synchronizing sequence for A by using the SCC method given in Figure 3, where for each $A_i = (S_i, \Sigma, D_i, \delta_i)$, we use Greedy and SynchroP to find S'_i-synchronizing sequence as explained in Section 3.

Figure 8 shows the performance improvements in terms of time and sequence length due to the SCC method over the original Greedy algorithm. We measured the ratios of the execution time and sequence length of original Greedy to those of Greedy with SCC method and report the averages of these metrics. The first three columns in each block of Figure 8 present the improvements over time, i.e., speedups, whereas the last three columns do the same for average sequence length. As the figure shows, for all DAG types, the SCC method is more than 6x faster than the original Greedy for automata with 8 SCCs. For binary-tree DAGs, the speedups look more; however, one should note that for tree DAGs, the results are given for 3,7 and 15 SCCs (instead of 2, 4 and 8). Hence for tree DAGs, the proposed method yields 12x speedup for automata with 15 SCCs. This also confirms that the speedups tend to increase with the number of SCCs. As the figure shows, the SCC method reduces the sequence lengths on average. Furthermore, similar to the average execution time, the amount of the reduction on sequence lengths tends to increase with the number of SCCs. We present a more detailed result set in Tables 1-6.

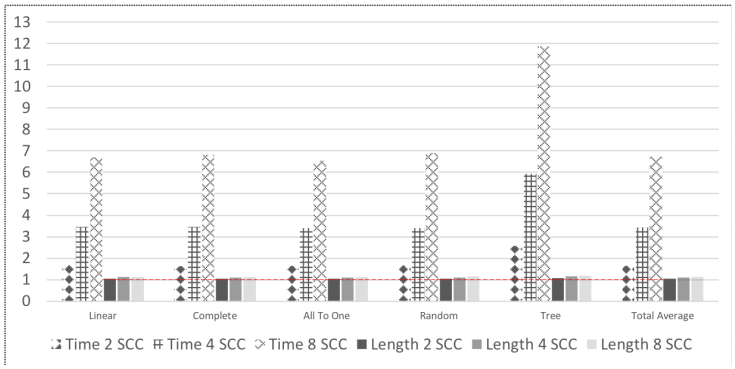

Figure 8: Greedy/SCC Method Time Ratio (Speedup) and Sequence Length Ratio Results for Automata with 2, 4 and 8 SCC's and 5 DAG types (3,7 and 15 SCC's for Tree DAGs).

Compared to Greedy, SynchroP is a slower heuristic. Hence, we expect that the impact of the SCC method over SynchroP will be more. Figure 9 confirms our expectations. As the figure shows, the proposed method can make SynchroP around 10x, 100x and 1000x faster on automata with 2, 4 and 8 SCCs, respectively. For tree DAGs, the improvement is 10000x on average for 15 SCCs. Thus, similar to the Greedy, one can say that the improvement on runtime tends to increase with the number of SCCs. However, it slightly decreases when the number of inputs, i.e., the alphabet size, increases.

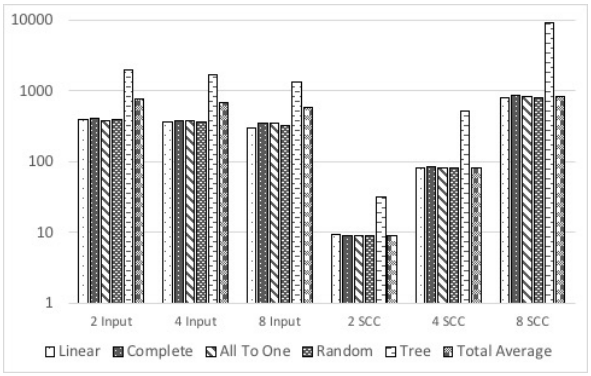

Figure 9: SynchroP/SCC Method Time Ratio (Speedup) Results for Automata with 2, 4 and 8 inputs and 5 DAG types in logarithmic scale (3,7 and 15 SCC's for Tree DAGs).

Although SynchroP is a slower heuristic, compared to Greedy, it is also much better in terms of sequence lengths. Hence, it should be harder to improve the length of the sequences. In fact, one can expect longer sequences for the SCC-based variant. In our experiments, for DAG types All-to-one, Linear, Complete, Tree and Random, the average ratios of the sequence lengths of original SynchroP and the SCC-based variant are 0.95, 1.04, 0.96, 0.95 and 0.98. Hence, the proposed method increases the sequences length only around 2.4% on average, and it improves the lengths by 4% for

weakly connected automata whose SCCs are connected like a binary tree. The detailed results on the sequence lengths for different parameter sets can be found in Tables 1-6.

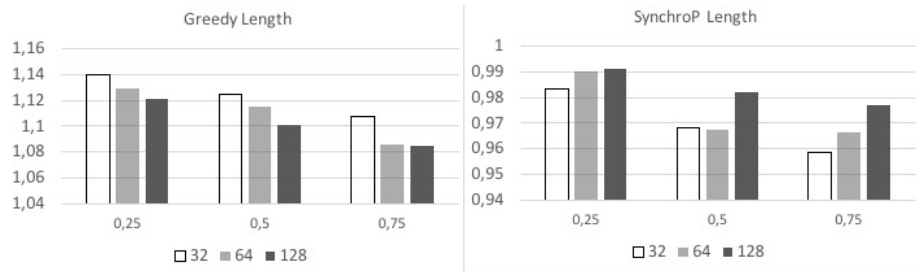

Figure 10: SynchroP/SCC Method and Greedy/SCC Method Sequence Length Ratio with Distribution Factors 0,25, 0,5, 0,75 and nSCCs 32, 64, 128**.**

To visualize the results on sequence lengths more clearly, Figure 10 presents the trends for varying distribution factors and number of SCCs used to generate the automata. As shown in the figure, for Greedy, the improvement on the sequence length decreases both with increasing number of states in each SCC (bottom legends in the figure) and increasing distribution factor. That being said, even in the worst pair of parameters the improvement is 8% on average. The results with SynchroP are different: when the number of SCCs increases, the SCC-based variant gets closer to original SynchroP. However, similar to Greedy, the distribution factor negatively affects the proposed method.

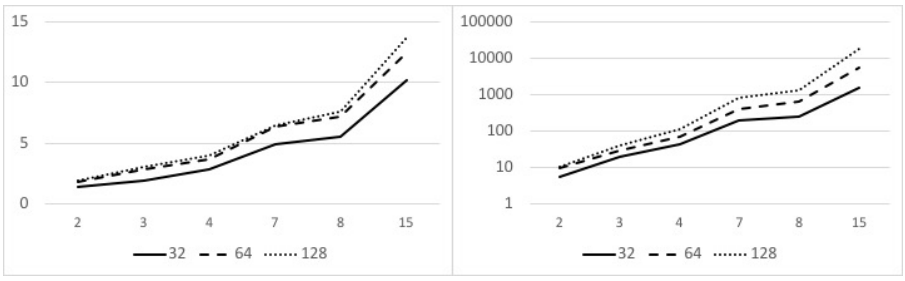

Figure 11: Greedy/SCC Method and SynchroP/SCC Method Time Ratio (Speedup) with 2, 3, 4, 7, 8 and 15 SCC's and 32, 64 and 128 nSCCs.

Figure 11 shows the trend on with different number of SCCs and nSCCs. Similar to the figures above, the speedups increase with the number of SCCs. They also increase with the number of states in a single SCC. This is expected since the automata get larger and the heuristics become more costly.

Table 1: Experimental results for All to One Automata

Number of States for each SCC	Number of SCCs	Number of Inputs	Greedy vs SCC Method		SynchroP vs SCC Method	
			Time Ratio	Length Ratio	Time Ratio	Length Ratio
32	2	2	0,74	1,04	4,87	1
		4	1,35	1,07	5,48	0,99
		8	1,5	1,08	5,57	0,99
	4	2	2,56	1,11	53,57	0,94
		4	2,97	1,14	47,89	0,94
		8	2,96	1,12	39,03	0,91
	8	2	5,32	1,11	318,62	0,94
		4	5,47	1,14	267,33	0,92
		8	5,4	1,15	209,87	0,84
64	2	2	1,78	1,02	10,22	0,98
		4	1,68	1,07	10,36	1,01
		8	1,91	1,06	7,9	1,01
	4	2	3,44	1,06	86,74	0,96
		4	3,63	1,11	77,65	0,96
		8	3,7	1,12	60,84	0,92
	8	2	6,93	1,06	851,08	0,93
		4	6,73	1,13	606,1	0,91
		8	7,16	1,14	542,36	0,86
128	2	2	1,96	1,04	9,72	0,99
		4	1,98	1,05	10,11	1,01
		8	1,93	1,06	9,94	1,01
	4	2	3,88	1,04	103,49	0,97
		4	3,8	1,13	113,04	0,95
		8	3,96	1,1	115,95	0,95
	8	2	7,46	1,08	1297,94	0,94
		4	7,1	1,16	1427,74	0,91
		8	7,63	1,12	1414,44	0,9

Table 2: Experimental results for Linear Automata

Number of States for each SCC	Number of SCCs	Number of Inputs	Greedy vs SCC Method		SynchroP vs SCC Method	
			Time Ratio	Length Ratio	Time Ratio	Length Ratio
32	2	2	1,09	1,04	6,07	1
		4	1,24	1,07	5,88	0,99
		8	1,98	1,08	5,7	0,99
	4	2	2,49	1,14	51,37	1,01
		4	2,69	1,17	43,92	1,02
		8	3,06	1,09	36,31	1,07
	8	2	4,93	1,18	337,16	1,03
		4	5,31	1,15	246,47	1,11
		8	5,93	1,09	190,13	1,14
64	2	2	1,89	1,02	10,86	0,98
		4	1,5	1,07	10,4	1,01
		8	1,84	1,06	7,86	1,01
	4	2	3,54	1,11	82,46	0,98
		4	3,56	1,14	76,65	1,06
		8	3,85	1,1	59,64	1,05
	8	2	6,69	1,17	898,92	1
		4	6,84	1,15	603,7	1,1
		8	7,51	1,07	532,77	1,14
128	2	2	1,89	1,04	9,88	0,99
		4	1,86	1,05	10,68	1,01
		8	1,93	1,06	9,6	1,01
	4	2	3,95	1,08	107,72	0,99
		4	3,72	1,14	116,89	1,04
		8	4,01	1,09	107,85	1,05
	8	2	7,22	1,15	1320,74	1,02
		4	6,86	1,16	1366,23	1,1
		8	8,19	1,07	1163,29	1,12

Table 3: Experimental results for Complete Automata

Number of States for each SCC	Number of SCCs	Number of Inputs	Greedy vs SCC Method		SynchroP vs SCC Method	
			Time Ratio	Length Ratio	Time Ratio	Length Ratio
32	2	2	1,48	1,04	5,26	1
		4	1,48	1,07	5,59	0,99
		8	1,4	1,08	5,87	0,99
	4	2	2,47	1,11	55,03	0,97
		4	3	1,14	48,5	0,99
		8	2,85	1,09	38,61	0,95
	8	2	5,43	1,11	335,4	0,92
		4	5,28	1,15	271,77	0,92
		8	5,76	1,12	215,82	0,87
64	2	2	1,6	1,02	10,53	0,98
		4	1,62	1,07	10,39	1,01
		8	1,78	1,06	7,74	1,01
	4	2	3,63	1,05	88,12	0,98
		4	3,68	1,12	78,49	0,98
		8	3,77	1,09	64,21	0,94
	8	2	7,28	1,11	920,73	0,94
		4	7,05	1,13	647,01	0,93
		8	7,45	1,11	633,61	0,89
128	2	2	1,91	1,04	9,62	0,99
		4	1,94	1,05	10,04	1,01
		8	1,9	1,06	10,1	1,01
	4	2	4,02	1,06	108,14	0,96
		4	3,83	1,13	119,44	0,99
		8	4	1,1	115,12	0,97
	8	2	7,52	1,11	1395,27	0,95
		4	7,3	1,16	1428,9	0,94
		8	8,01	1,12	1407,34	0,92

Table 4: Experimental results for Tree Automata

Number of States for each SCC	Number of SCCs	Number of Inputs	Greedy vs SCC Method		SynchroP vs SCC Method	
			Time Ratio	Length Ratio	Time Ratio	Length Ratio
32	3	2	1,67	1,07	19,11	0,97
		4	1,81	1,12	20,08	0,98
		8	2,12	1,11	17,82	0,93
	7	2	4,68	1,14	251,53	0,94
		4	4,89	1,16	210,9	0,94
		8	5,02	1,18	150,54	0,93
	15	2	9,73	1,17	1898,79	0,96
		4	10,04	1,22	1591,15	0,94
		8	10,58	1,2	1299,54	0,94
64	3	2	6,13	1,13	481,43	0,95
		4	5,95	1,18	446,86	0,96
		8	6,66	1,15	340,69	0,97
	7	2	6,13	1,13	481,43	0,95
		4	5,95	1,18	446,86	0,96
		8	6,66	1,15	340,69	0,97
	15	2	12,45	1,17	7360,79	0,96
		4	11,85	1,22	5763,36	0,95
		8	12,97	1,19	4320,21	0,94
128	3	2	2,99	1,06	40,28	0,95
		4	2,99	1,12	40,22	0,98
		8	3,11	1,08	40,56	0,97
	7	2	6,55	1,11	799,83	0,95
		4	6,23	1,2	861,5	0,98
		8	6,59	1,14	809,63	0,97
	15	2	13,38	1,13	21815,24	0,94
		4	12,53	1,23	20093,27	0,93
		8	14,54	1,16	16178,07	0,96

Table 5: Experimental results for Random Automata

Number of States for each SCC	Number of SCCs	Number of Inputs	Greedy vs SCC Method		SynchroP vs SCC Method	
			Time Ratio	Length Ratio	Time Ratio	Length Ratio
32	2	2	1,21	1,04	5,61	1
		4	1,54	1,07	5,77	0,99
		8	1,45	1,08	5,73	0,99
	4	2	2,48	1,11	54,68	0,96
		4	2,7	1,13	45,33	0,97
		8	2,99	1,1	36,38	0,98
	8	2	5,41	1,17	334,1	0,95
		4	5,81	1,15	258,82	0,97
		8	5,97	1,16	199,64	0,96
64	2	2	1,64	1,02	10,22	0,98
		4	1,78	1,07	10,36	1,01
		8	1,8	1,06	7,9	1,01
	4	2	3,57	1,09	86,74	0,98
		4	3,43	1,14	77,65	0,99
		8	3,8	1,11	60,84	0,98
	8	2	6,96	1,13	851,08	0,95
		4	6,9	1,15	606,1	0,98
		8	7,7	1,14	542,36	0,97
128	2	2	1,98	1,04	9,55	0,99
		4	1,9	1,05	10,42	1,01
		8	1,91	1,06	9,91	1,01
	4	2	3,88	1,07	107,13	0,98
		4	3,73	1,13	115,24	1
		8	4,06	1,1	111,98	1
	8	2	7,25	1,12	1326,64	0,97
		4	7,28	1,17	1392,2	1
		8	8,23	1,14	1284,89	0,99

6 Conclusions

The SCC method can be used with any synchronizing heuristic to make it run faster on weakly connected automata. In case of Greedy, it can also find shorter reset sequences in shorter time compared to the application of Greedy directly. Compared to Greedy, SynchroP is a slower heuristic, which finds shorter reset sequences. With the proposed SCC method, one can use SynchroP to generate slightly longer (still shorter compared to Greedy) sequences and can be thousands of times faster.

We were inspired by the following observation while designing the SCC-based method: The Greedy heuristic requires $O(n^3)$ and SynchroP requires $O(n^5)$ time where n is the number of states in the automata. Therefore, if one can divide an automaton A into pieces (components) in a way that the synchronizing sequence of A can be constructed from the synchronizing sequences obtained for these pieces, this approach can result in considerable time savings. In this work, we suggest that these "pieces", whose synchronizing sequences can be combined easily to form a synchronizing sequence of the original automaton, are the SCCs of a weakly connected automaton. If there are k SCCs with equal sizes, complexity of Greedy and SynchroP becomes $O(k(\frac{n}{k})^3)$ $O(k\,(\frac{n}{k})^5)$, respectively. Our experiments show that, the running times improve as expected. Of course, the SCC method works only for weakly connected automata and the effectiveness of the method depends on the number and the size of the SCCs, as displayed by the experiments.

For future work, one direction is to improve our synchronizing sequence lengths for the SCC method when used with SynchroP. The direct application of SynchroP algorithm to an automaton performs a global analysis compared to the local analysis performed when each SCC is analyzed separately by our SCC method. Another direction of research can be to find other decompositions for an automaton.

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

References

1. Cirisci, B., Kahraman, M. K., Yildirimoglu, C. U., Kaya, K., Yenigun, H., 2018. Using Structure of Automata for Faster Synchronizing Heuristics. In Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, 544-551.
2. Chow, T.S., 1978. Testing software design modelled by finite state machines. IEEE Transactions on Software Engineering, 4:178-187.
3. Eppstein, D., 1990. Reset sequences for monotonic automata. SIAM J. Comput. 19 (3), 500 - 510.
4. Hierons, R.M., Ural, H. 2006. Optimizing the length of checking sequences. IEEE Transactions on Computers. 55(5): 618-629.
5. Jourdan, G.V., Ural, H., Yenigün, H., 2015. Reduced checking sequences using unreliable reset, Information Processing Letters, 115(5), 532-535.
6. Kudlacik, R., Roman, A., Wagner, H., 2012. Effective synchronizing algorithms. Expert Systems with Applications 39 (14), 11746-11757.

7. Lee, D., Yannakakis, M., 1996. Principles and methods of testing finite state machines-a survey. Proceedings of The IEEE, 84(8), 1090-1123.
8. Roman, A., Szykula, M., 2015. Forward and backward synchronizing algorithms. Expert Systems with Applications 42 (24), 9512-9527.
9. Rystsov, I., 1997. Reset words for commutative and solvable automata. Theoretical Computer Science 172 (1-2), 273–279.
10. Trahtman, A. N., 2004. Some results of implemented algorithms of synchronization. In: 10th Journees Montoises d'Inform.
11. Volkov, M.V., 2008. Synchronizing automata and the Cerny conjecture. In Proceedings of the 3rd International Conference on Language and Automata Theory and Applications, LATA' 08, 11–27, 2008.
12. Volkov, M.V., 2009. Synchronizing automata preserving a chain of partial orders. Theoretical Computer Science 410 (37), 3513–3519.