Using Structure of Automata for Faster Synchronizing Heuristics

Berk Cirisci, Muhammed Kerem Kahraman, Cagri Uluc Yildirimoglu, Kamer Kaya and Husnu Yenigun

Computer Science and Engineering, Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey

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 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 (Chow, 1978; Lee and Yannakakis, 1996; Hierons and Ural, 2006). 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 (Jourdan et al., 2015). For more examples of application areas of synchronizing sequences and for an overview of the theoretical results related to synchronizing sequences please see (Volkov, 2008).

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 (Eppstein, 1990). Therefore, heuristic algorithms, known as synchronizing heuristics, are used to find short synchronizing sequences. Among such heuristics are Greedy (Eppstein, 1990), Cycle (Trahtman, 2004), SynchroP (Roman, 2009), and SynchroPL (Kudlacik et al., 2012). 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 \( A \) is called strongly connected if every state is reachable from every other state by using at least one sequence of inputs. Otherwise, \( A \) is called non-strongly connected and in this case \( A \) can be represented as a set of strongly connected automata. These automata are called as strongly connected components (SCCs) of \( A \).

In this paper, given a non-strongly connected automaton \( A \), we suggest a method to build a synchronizing sequence for \( A \) by using the synchronizing sequences of the SCCs of \( A \). We considered the application of Greedy and SynchroP algorithms directly to an automaton, and to SCCs of the automaton. We observe that, the suggested methods improve the running time greatly, without
extending the length of the synchronizing sequences much.

The remaining part 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, \( \Sigma \) 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 \( \Sigma^* \) is called a sequence. For a sequence \( w \in \Sigma^* \), \( |w| \) denotes the length of \( w \), and \( \varepsilon \) is the empty sequence of length 0. For a complete automaton, we extend the transition function \( \delta \) 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 \( \delta(C, w) = \{(\delta(s, w)) s \in C\} \).

For a set of states \( C \subseteq S \), let \( C^2 = \{(s, s') s, s' \in C\} \) be the set of all multiset elements of cardinality 2 with elements from \( C \). \( C^2 \) 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 \( C^s \) and \( C^d \) 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. Note that, for an s-pair \( \{s, s\} \), every sequence (including \( \varepsilon \) ) is a merging sequence. A sequence \( w \) is called an \( S \)-synchronizing sequence for an automaton \( A = (S, \Sigma, \Sigma \times \Sigma, \delta) \) and a subset of states \( S' \subseteq S \) 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 simply called synchronizing if there exists a synchronizing sequence for \( A \).

In this paper, we only consider synchronizing automata. As shown by Eppstein (1990), 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 write \( \delta^{-1}(s, x) \) to denote the set of states with a transition to state \( s \) with letter \( x \), i.e., \( \delta^{-1}(s, x) = \{s' \in S | \delta(s', x) = s\} \). We also define \( \delta^{-1}(\{s, s'\}, x) = \{(p, p') | p \in \delta^{-1}(s, x) \land p' \in \delta^{-1}(s', x)\} \).

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 \( D \subseteq S \times \Sigma \), 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) \( \forall (s, x) \in D, \delta(s, x) = \delta(s, x) = \delta(s, x) = \delta(s, x) = \delta(s, x) \).

Intuitively, the states of \( B \) consist of a subset of states of \( A \). Every transition in \( A \) from a state in \( B \) to a state in \( B \) 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 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 strongly connected component of the 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.

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 \( S_i \) 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.
3 SYNCHRONIZING SEQUENCES FOR NON-STRONGLY CONNECTED AUTOMATA

Consider an automaton $A = (S, \Sigma, S\times \Sigma, \delta)$ and its SCC decomposition $\{A_1, A_2, \ldots, A_t\}$.

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

Proof: If there are two distinct sink components $A_i$ and $A_j$ of $A$, a state $s_i$ in $A_i$ and a state $s_j$ in $A_j$ can never be merged. If $A_i$ is the only sink component of $A$ and $A_j$ is not synchronizing, $A$ is not synchronizing as well.

Let $A = (S, \Sigma, S\times \Sigma, \delta)$ be an automaton and $\{A_1, A_2, \ldots, A_t\}$ be the SCCs of $A$. We consider the SCCs of $A$ (topologically) sorted as $\{A_1, A_2, \ldots, A_t\}$ such that for any $1 \leq i < j \leq k$, there do not exist $s_i \in S_{j}$, $s_j \in S_{j}$, $w \in \Sigma^*$ where $\delta(s_i, w) = s_j$. Note that in this case $A_i$ must be a sink component and we have the following result.

Lemma 2: Let $\{A_1, A_2, \ldots, A_t\}$ be a topologically sorted SCCs of an automaton $A = (S, \Sigma, S\times \Sigma, \delta)$, where $A_i = (S_i, \Sigma, D_i, \delta_i)$, $1 \leq i \leq k$. For any sequence $w \in \Sigma^*$ and for a state $s \in S_i$, $1 \leq i \leq k$, 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. If $A_i$ is a partial automaton, then the completion $A'_i$ of $A_i$ is a synchronizing automaton.

Proof: Since $A_i$ is an SCC, all states can be reached from other states in $A_i$. Also, we know that star state is a state that can reach to only itself. When we complete $A_i$ with a star state, every state can reach the star state and star state can’t reach to other state then itself so that means other states should unite in star state eventually and makes $A_i$ a synchronizing automaton.

3.1 An Initial Approach to Use SCCs

We now explain an initial idea to form a synchronizing sequence for an automaton $A$ by using synchronizing sequences of the SCCs of $A$. Let $A = (S, \Sigma, S\times \Sigma, \delta)$ be an automaton and $\{A_1, A_2, \ldots, A_t\}$ be the topologically sorted SCCs of $A$, where $A_i = (S_i, \Sigma, D_i, \delta_i)$, $1 \leq i \leq k$. Let $\beta_i$ be a synchronizing sequence for the completion $A'_i$ of $A_i = (S_i, \Sigma, D_i, \delta_i)$. Note that based on Lemma 2 one can always find a synchronizing sequence for $A_i$, $1 \leq i \leq k$. Let $\beta_h$ be a synchronizing sequence for $A_h$. Lemma 1 suggests that $A_h$ 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 $0 \leq i < k$ we have

$\delta(S, \beta_1 \beta_2 \ldots \beta_i) \subseteq (S_{i+1} \cup S_{i+2} \cup \ldots \cup S_k)$

Proof: By induction, where the base case $i = 0$ holds trivially. Assume that the claim holds for $i-1$, i.e. $\delta(S, \beta_1 \beta_2 \ldots \beta_{i-1}) \subseteq (S_i \cup S_{i+1} \cup \ldots \cup S_k)$. For a state $s \in \delta(S, \beta_1 \beta_2 \ldots \beta_{i-1})$ such that $s \in (S_{i+1} \cup S_{i+2} \cup \ldots \cup S_k)$, then $\delta(s, \beta_i)$ will also belong to $(S_{i+1} \cup S_{i+2} \cup \ldots \cup S_k)$ based on Lemma 2. Hence it remains to show that for any state $s \in \delta(S, \beta_1 \beta_2 \ldots \beta_{i-1})$ such that $s \in S_i$, $\delta(s, \beta_i) \in (S_{i+1} \cup S_{i+2} \cup \ldots \cup S_k)$.
δ(s, βs) is not in Si. The sequence β is a synchronizing sequence for the completion A' of SCC Ai. Since the star state of A' is the only state in which the states of A' can be synchronised, we must have δ(S, βs) = {*}. Note that the star state in A' represents the states SX for A. Hence the sequence β is in fact a sequence that pushes all the states in S to the states in the other components, i.e., δ(S, βs) = ∅. This implies that for a state s ∈ δ(S, βs) = ∅. Finally, we can state the following result.

**Theorem 5:** Let (A1, A2, ..., Ak) be a topologically sorted SCCs of an automaton A = (S, Σ, D, δ), where

A = (S, Σ, D, δ); 1 ≤ i ≤ k. Let β be a synchronising sequence for the completion A' of Ai, 1 ≤ i < k, and let βs be a synchronising sequence for Ai. The sequence βs is a synchronising sequence for A.

**Proof:** δ(S, βs) ∩ δ(S, βs) = ∅, using Lemma 4. Since βs is a synchronising sequence for Ai, δ(S, βs) is singleton. Combining these two results, we have

δ(S, βs) ∩ δ(S, βs) = δ(S, βs) ∩ δ(S, βs) as singleton as well.

### 3.2 An Improvement on the Initial Approach

Theorem 5 shows an easy way for constructing a synchronising sequence for an automaton A based on its SCCs. As one may notice, through the length of the sequence to be constructed can be reduced based on the following observation. Consider a sequence βs for some 1 ≤ i ≤ k, used in the sequence βs...βs,βs. The sequence βs is constructed to push all the states in Ai out of the component Ai. However, the sequence βs...βs,βs applied before βs can already push some states in Ai out of Ai. On the other hand, the sequence βs...βs,βs can also move some of the states in the components A1, A2, ..., Ai to a state in Ai. Therefore, a more careful approach can be taken considering which states in Ai must really be moved out of Ai when constructing the sequence to handle the component Ai.

To take this into account, we define the following sequences recursively. For the base cases we define α0 = ε and α0 = ε. For 1 ≤ i < k, let S'i = S; ∩ δ(S, σi) and let αi be a S'i-synchronising sequence for A'i. For 1 ≤ i < k, let σi = σi,1 αi.

**Theorem 6:** Let S'i = S; ∩ δ(S, σi) and αi be a S'i-synchronising sequence for A'i. Then αi,1 αi is a synchronising sequence for A.

**Proof:** σk,1 is a synchronising sequence for A1 ∪ A2 ∪ A3 ∪ ... ∪ Ak and we know that they are synchronised in the star state of Ak, which represents the states that are outside of Ak. These states belong to Ai because Ai is ahead of Ai in topological sort and it is the only SCC left so we can say that δ(S, σk,1) ⊆ Sk. In other words, σk,1 leaves us with active states Sk ⊆ S; since αk synchronises all the states of Sk, σk,1 αk is a synchronising sequence for A.

Based on Theorem 6, the algorithm given in Figure 3 can be used to construct a synchronising sequence for an automaton A.

**Input:** An automaton A = (S, Σ, D, δ)

**Output:** A synchronising sequence for A

C = S; // All states are active initially
Γ = ε; // Γ: synchron. sequence to be constructed, initially // empty

foreach i in {1, 2, ..., k} do
    // Consider Ai = (Si, Σ, Di, δ)
    S'i = S; ∩ δ(S, σi) // find active states of Ai
    Ti = Heuristic(A'i, S'i); // find S'i sync. sequence of completion A'i of Ai
    Γ = Γ Ti; // append Ti to sync. seq.
    C = δ(C, Γ); // Update active states
return Γ;

Figure 3: SCC algorithm to compute synchronising sequences.

Note that in the algorithm given in Figure 3, any synchronizing heuristic can be used to compute Γ. In the next section, we explain two different algorithms from the literature that we used in our experiments.

### 4 SYNCHRONIZING HEURISTICS

As noted in Section 1, there are various synchronising heuristics. In this paper, we considered and experimented with two of these heuristics, Greedy and SynchroP. Both Greedy and SynchroP heuristics have two phases. Phase 1 is
common in these heuristics and given as Algorithm 1 below. In Phase 1, a shortest merging sequence \( \tau(i,j) \) for each \( (i,j) \in S^2 \) is computed by using a breadth first search. Note that \( \tau(i,j) \) is not unique.

**Input:** An automaton \( A = (S, \Sigma, D, \delta) \)

**Output:** A merging sequence for all \( (i,j) \in S^2 \)

let \( Q \) be an initially empty queue 
// \( Q \): BFS frontier

\[ P = \emptyset \] // \( P \): keeps the set of nodes 
// in the BFS forest
// constructed so far

foreach \( (i,j) \in S^2 \), do

push \( (i,j) \) onto \( Q \)

insert \( (i,j) \) into \( P \)

set \( \tau(i,j) = i \); 

while \( P \neq S^2 \) do

\( (i,j) = \text{pop next item from } Q \); 

do 

foreach \( k,l \in \delta^{-1}(i,j), x \) do 

if \( (k,l) \notin P \) then 

\( \tau(k,l) = x \cdot \tau(i,j) \); 

push \( (k,l) \) onto \( Q \); 

\( P = P \cup \{(k,l)\} \); 

Figure 4: Phase 1 of Greedy and SynchroP.

Algorithm 1 performs a breadth first search (BFS), and therefore constructs a BFS forest, rooted at \( s \)-pairs \( (i,j) \in S^n \), where these \( s \)-pairs are the nodes at level 0 of the forest. A \( d \)-pair \( (i,j) \) appears at level \( k \) of the BFS forest if \( |\tau(i,j)| = k \).

Algorithm 1 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 as Algorithm 2 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 \( O(n^2) \) time, but using the information precomputed by the intermediate stage between Phase 1 and Phase 2, line 6 can be handled in \( O(n) \) time. Therefore, Phase 2 of Greedy requires \( O(n^3) \) time. Note that Algorithm 2 finds an \( S \)-synchronizing sequence for a given complete automaton \( A = (S, \Sigma, S, \Sigma, \delta) \). However, for our purposes we need to find an \( S' \)-synchronizing sequence for a given subset \( S' \subseteq S \) of states.

**Input:** An automaton \( A = (S, \Sigma, D, \delta) \), \( \tau(i,j) \) for all \( (i,j) \in S^2 \), \( S' \) to be synchronized 

**Output:** An \( S' \)-synchronizing sequence \( \Gamma \) for \( A \)

\( C = S' \) // \( C \): current state set
\( \Gamma = \varepsilon \) // \( \Gamma \): synthes. sequence to be constructed, initially // empty 

while \( |C| > 1 \) do // still not a singleton

\( (i,j) = \arg\min_{k,l} \{d(i,j) | \delta(k,l) \} \); 
// decide the \( d \)-pair to be merged

\( \Gamma = \tau(i,j) \); // append \( \tau(i,j) \)
// to the // synchronizing // sequence

\( C = \delta(C, \tau(i,j)) \); // update current // state set // with \( \tau(i,j) \)

Figure 5: Phase 2 of Greedy.

**4.2 The SynchroP Heuristic**

Similar to the second phase of Greedy, the second phase of SynchroP also constructs a synchronizing sequence iteratively. The algorithms keep 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 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 \( \phi(C) \) of \( C \) be defined as

\[
\phi(C) = \sum_{i,j \in C} |\tau(i,j)|
\]

\( \phi(C) \) 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 sequence would be required to bring \( C \) to a singleton set.

During the iterations of SynchroP, the selection of \( (i,j) \in C \) 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 Algorithm 2. Like in Greedy with SCC Method, we also use a slightly modified version of the second phase of SynchroP algorithm to find \( S' \)-synchronizing sequence.
Input: An automaton $A = (S, \Sigma, D, \delta)$, 
$\tau(i,j)$ for all $\{i,j\} \in S^2$,
$S'$ to be synchronized
Output: An $S'$-synch. sequence $\Gamma$ for $A$

$C = S'$ // C: current state set
$\Gamma = \epsilon$ // $\Gamma$: synchronizing sequence to // be constructed, initially // empty

while $|C| > 1$ do // still not a // singleton
    minCost = $\infty$
    foreach d-pair $(i,j) \in C \times D$ do
        thisPairCost = $\frac{1}{|S'|} \sum_{s \in S'} \delta(C, \tau(i,j))$
        if thisPairCost < minCost then
            minCost = thisPairCost
            $\tau' = \tau(i,j)$

    $\Gamma = \Gamma \tau'$; // append $\tau'$ to the // synchron. sequence
    $C = \delta(C, \tau')$; // update current // state set // with $\tau'$

Figure 6: Phase 2 of SynchroP.

5 EXPERIMENTAL RESULTS

The experiments were performed on a machine with Intel Xeon E5-1650 CPU and 16GB 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 method suggested in this paper, we generated random automata with $n \in \{256, 512, 1024\}$ states, $p \in \{2, 4, 8\}$ inputs, $k \in \{2, 4, 8\}$ SCCs in the following way. To construct a random automaton $A$ with a given number of states $n$, number of inputs $p$, and number of SCCs $k$, we first construct $k$ different automata $A_1, A_2, \ldots, A_k$ where each $A_i$ is strongly connected, has $nk$ states and $p$ inputs. To construct $A_i$, we consider each state $s$ in $A_i$ and each input $x$, and assign $\delta(s,x)$ to be one of the states in an automaton $A_{s1}, A_{s2}, \ldots, A_s$. For each $n-p-k$ combination we created 50 random automata. The results given later in this section are the average of these 50 automata.

For an automaton $A = (S, \Sigma, S \times \Sigma, \delta)$ with $n$ states, $p$ inputs and $k$ SCCs $(A_1, A_2, \ldots, A_k)$ where $A_i = (S_i, \Sigma, D_i, \delta_i), 1 \leq i \leq k$, 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 Algorithm given in Figure 3, where for each $A_i = (S_i, \Sigma, D_i, \delta_i)$ we use Greedy and SynchroP algorithms to find $S'_i$-synchronizing sequence as explained in Section 3.

Table 1 gives the running time and the synchronizing sequence length for the direct application of Greedy and SynchroP compared to the SCC method suggested in this paper.

As expected, the running time is improved in all the cases. The speed-up values (i.e. the time required for the direct application of Greedy/SynchroP divided by the time required for the application of SCC method using Greedy/SynchroP) do not change much based on the number of inputs of the automata. However, the number of states and the number of SCCs of the automata are very important factors for the speed-up values. Figure 7 and Figure 8 display the speed-up values obtained in a more explicit way. For the time performance, the SCC method becomes more effective as the size of the automaton and the number of SCCs increase.

For the length of the synchronizing sequences found, the SCC method finds even shorter sequences (5% shorter on the average) compared to the direct application of Greedy. Although the direct application of SynchroP yields shorter synchronizing sequences in general, the increase in the length is not large (3% longer on the average).

6 CONCLUSIONS

The SCC-based method suggested in this paper is a method that can be used with any synchronizing heuristic to make it run faster on non-strongly connected automata. In case of Greedy, it can also find shorter reset sequences in shorter time compared to the application of Greedy directly. SynchroP is a method which typically finds shorter reset sequences compared to Greedy but it takes more time. With our method, we can use SynchroP to find shorter reset sequences and also SCC method will not take more time than the direct application of Greedy.
Table 1: Experimental results for Greedy and SynchroP.

<table>
<thead>
<tr>
<th>Number of States</th>
<th>Number of SCCs</th>
<th>Inputs</th>
<th>Greedy with SCC Method</th>
<th>Greedy</th>
<th>SynchroP with SCC Method</th>
<th>SynchroP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Length</td>
<td>Time</td>
<td>Length</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ms)</td>
<td></td>
<td>(ms)</td>
<td></td>
<td>(ms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3.12</td>
<td>35.84</td>
<td>5.71</td>
<td>38.08</td>
<td>60.39</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.06</td>
<td>20.66</td>
<td>7.66</td>
<td>22.22</td>
<td>46.88</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6.72</td>
<td>29.64</td>
<td>12.72</td>
<td>29.94</td>
<td>53.68</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
<td>2.02</td>
<td>31.96</td>
<td>6.617</td>
<td>36.47</td>
<td>5.26</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.38</td>
<td>22.26</td>
<td>8.66</td>
<td>23.72</td>
<td>5.78</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3.1</td>
<td>23.9</td>
<td>11.82</td>
<td>23.76</td>
<td>7.92</td>
</tr>
<tr>
<td>512</td>
<td>2</td>
<td>13.8</td>
<td>44.88</td>
<td>23.52</td>
<td>46.18</td>
<td>472.28</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>19.96</td>
<td>30.5</td>
<td>35.24</td>
<td>31.1</td>
<td>405.64</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>27.88</td>
<td>45.36</td>
<td>48.66</td>
<td>46.2</td>
<td>485.92</td>
</tr>
<tr>
<td>1024</td>
<td>2</td>
<td>6.69</td>
<td>40.35</td>
<td>23.54</td>
<td>42.88</td>
<td>64.38</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8.46</td>
<td>23.7</td>
<td>33.08</td>
<td>24.74</td>
<td>50.36</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>14.48</td>
<td>34.42</td>
<td>55.7</td>
<td>34.18</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.6</td>
<td>23.7</td>
<td>35.542</td>
<td>25.68</td>
<td>8.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.56</td>
<td>25.8</td>
<td>59.34</td>
<td>26.58</td>
<td>11.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>23.64</td>
<td>172.18</td>
<td>35.12</td>
<td>71.38</td>
</tr>
</tbody>
</table>

The time improvements we obtain by using the SCC method are expected. Greedy requires $O(n^3)$ and SynchroP requires $O(n^5)$ time where $n$ is the number of states. Therefore, if one can divide the automaton into pieces (components) in one way or the other, and construct a synchronizing sequence from the synchronizing sequences obtained for these pieces, this approach would result in considerable time savings. In this paper, we suggest that these “pieces” can be the strongly connected components.
Figure 7: SynchroP/SCC Method Time Ratio (Speedup) Results of Automata’s with 256,512,1024 States and 2,4,8 SCC’s.

Figure 8: Greedy/SCC Method Time Ratio (Speedup) Results of Automata’s with 256,512,1024 States and 2,4,8 SCC’s.

of the automaton. Because of the reasons above, SCC method can make every heuristic faster as shown in this paper. If there are $k$ strongly connected components with equal sizes, complexity of Greedy and SynchroP becomes $O(k \left( \frac{n}{k} \right)^5)$ and $O(k \left( \frac{n}{k} \right)^3)$, respectively. Obviously, these are much faster running times compared to original heuristics. In practice, the running times differ as expected.

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 performs a global analysis compared to the local analysis performed when each strongly connected component is analyzed separately by our SCC method. Another direction of research is to use the SCC method with other synchronizing heuristics and to extend the experiments to study the effects of aspects like states, inputs, number of SCCs, and also the relative size of SCCs, a factor which we did not take into account in the experiments performed in this paper.

ACKNOWLEDGMENTS

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

REFERENCES


