Lookahead-Based Approaches for Minimizing Adaptive Distinguishing Sequences

Uraz Türker, Tonguç Ünlüyurt, Hüsnü Yenigün

Lookahead-based Approaches for Minimizing Adaptive Distinguishing Sequences

Uraz Cengiz Türker[1], Tonguç Ünlüyurt[1], and Hüsnü Yenigün[1]

Sabanci University, Orhanli, Tuzla, 34956, Istanbul, TURKEY
{urazc,tonguc,yenigun}@sabanciuniv.edu

Abstract. For Finite State Machine (FSM) based testing, it has been shown that the use of shorter Adaptive Distinguishing Sequences (ADS) yields shorter test sequences. It is also known, on the other hand, that constructing a minimum cost ADS is an NP-hard problem and it is NP-hard to approximate. In this paper, we introduce a lookahead-based greedy algorithm to construct reduced ADSs for FSMs. The greedy algorithm inspects a search space to make a decision. The size of the search space is adjustable, allowing a trade-off between the quality and the computation time. We analyse the performance of the approach on randomly generated FSMs by comparing the ADSs constructed by our algorithm with the ADSs that are computed by the existing algorithms.

Keywords: Finite State Machines; Adaptive Distinguishing Sequences; greedy algorithms.

1 Introduction

Finite State Machines (FSMs) have been used to model reactive systems [1–6]. A number of techniques, which operate on FSM models of such systems, automatically generate test sequences [1, 7–14]. In order to reduce the cost of testing, shorter test sequences are preferrable and there is an increasing number of studies in the literature in this direction [13, 15–19].

State identification sequences [8, 9, 20] play a central role in the design of these test sequences and a large portion of these test sequences consists of state identification sequences. Many techniques for constructing test sequences use *Distinguishing Sequences* (DSs) as state identification sequences. There exist polynomial time algorithms that generate test sequences when a DS is given. Moreover, the length of the test sequence is relatively short when designed with a DS [15, 16, 21–23][1].

A distinguishing sequence can be preset or adaptive. If the input sequence is known before the experiment, then it is a *Preset Distinguishing Sequence* (PDS). If the input symbol to be applied is decided after observing the response of the FSM to the previous input, then it is an *Adaptive Distinguishing Sequence* (also

[1] Although the upper bound on PDS length is exponential, test generation takes polynomial time if there is a known PDS.

known as a *Distinguishing Set*) [21]. Therefore an ADS is essentially a decision tree. The internal nodes of the tree are labeled by input symbols, the edges are labeled by output symbols where the edges emanating from a common node have different labels, and the leaves are labeled by states.

Using an ADS rather than a PDS is advantageous due to the following. Lee and Yannakakis show that checking the existence and computing a PDS is a PSPACE-complete problem. On the other hand, for a given FSM M with n states and m inputs, the existence of an ADS can be decided in time $O(mn \log n)$, and if exists, an ADS can be constructed in time $O(mn^2)$ by using the algorithm presented by Lee and Yannakakis (which we refer to as *LY Algorithm* in this work) [23].

It was proven by Sokolovskii that if an FSM M has an ADS, then an ADS for M with height no greater than $\pi^2 n^2/12$ exists [24]. Later Lee and Yannakakis reduced this bound to $n(n-1)/2$ and they showed that this bound is tight [23]. Kushik et al. present an algorithm for constructing ADSs for nondeterministic observable FSMs [25]. Since the class of deterministic FSMs is a subclass of nondeterministic observable FSMs, naturally the algorithm can also be used to construct ADS for deterministic FSMs.

Although Lee and Yannakakis have proven a tight upper bound on the height of an ADS, there is no work attempting to construct reduced size ADSs, other than the expontential time exhaustive algorithms [20, 26]. Recently in [27], Türker and Yenigün showed the potential enhancements of using reduced cost ADSs on the length of test sequences. They also examined the computational complexity of constructing minimum cost ADSs, where the "cost" of an ADS is defined as (i) the height of the ADS, and (ii) the sum of lengths of all root–to–leaf paths in the ADS (which is called *the external path length* of the ADS). They showed that constructing a minimum ADS with respect to these cost metrics are NP-complete and NP-hard to approximate. Thus, to construct reduced size ADSs, heuristic algorithms are needed.

In this paper we present a lookahead-based method for constructing reduced ADSs. The proposed lookahead search methodology uses a tree structure (called EST) to construct reduced size ADSs. The construction of EST is guided by different heuristics based on the objective, such as minimizing the height or the external path length of the ADS. The EST is potentially an infinite tree but the proposed method constructs and evaluates the EST greedily, limiting the size of the EST. By choosing how far one wants to lookahead during the construction of an ADS, a trade off between the time to construct an ADS and the cost of the ADS is possible. We evaluate the proposed method by performing experimental studies on randomly generated FSMs. The results indicate that the proposed method produces better results than LY algorithm, possibly at the expense of more computation time.

In the rest of this paper we first give the definitions and notation that will be used in this paper. We then present the details of the algorithm to construct reduced ADSs. Finally, we present the results of the experiments, and conclude by some discussions and giving some future directions of research.

2 Preliminaries

2.1 Finite State Machines

An FSM (or a Mealy machine) M is defined by a tuple $M = (S, X, Y, \delta, \lambda)$ where S is a finite set of states, X and Y are finite sets of input and output symbols, $\delta : S \times X \to S$ is the state transition function, and $\lambda : S \times X \to Y$ is the output function. If M is at a state $s \in S$, and if an input $x \in X$ is applied, M moves to the state $s' = \delta(s, x)$ and produces the output $y = \lambda(s, x)$. Note that, since the transitions are defined by a function (rather than a relation), M is *deterministic*. When δ and λ are total functions, M is said to be *completely specified*.

We use juxtaposition to denote concatenation. For example, $w = x_1 x_2 \dots x_k$, where $x_i \in X$, $1 \leq i \leq k$, denotes an input sequence. An input/output sequence consists of a sequence of input/output pairs of the form $x_1/y_1 x_2/y_2 \dots x_m/y_m$. The symbol ϵ is used to denote the empty sequence. The transition function and the output function can be extended to sequences of inputs. By abusing the notation, we will use δ and λ for the extended functions. These extensions are defined as follows (where $x \in X$ and $w \in X^\star$): $\delta(s, \epsilon) = s$ and $\delta(s, xw) = \delta(\delta(s, x), w)$; $\lambda(s, \epsilon) = \epsilon$ and $\lambda(s, xw) = \lambda(s, x)\lambda(\delta(s, x), w)$. Two states $s, s' \in S$ are said to be *equivalent* if for all input sequences $w \in X^\star, \lambda(s, w) = \lambda(s', w)$. Otherwise, if there exists an input sequence $w \in X^\star$ such that $\lambda(s, w) \neq \lambda(s', w)$, then s and s' are said to be *distinguishable*. An FSM M is *minimal* if the states of M are pairwise distinguishable. In Figure 1, an example FSM is given which is deterministic, completely specified and minimal.

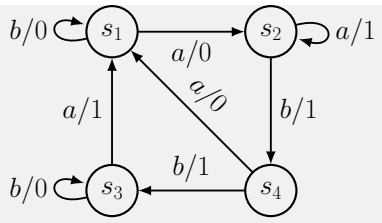

Fig. 1: An example deterministic, completely specified and minimal FSM M

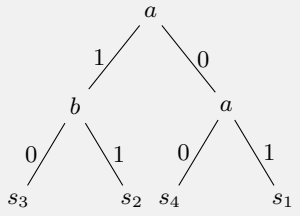

Fig. 2: The ADS $\mathcal{A}$ of FSM M in Figure 1.

A subset of states $B \subseteq S$ is called a *block*. We also extend δ and λ to a block as follows: For a block B and an input sequence w, $\delta(B, w) = \{\delta(s, w) \mid s \in B\}$ and $\lambda(B, w) = \{\lambda(s, w) \mid s \in B\}$. An input symbol x is said to be *valid* for a block B, for all $s, s' \in B$ such that $s \neq s'$, $\lambda(s, x) = \lambda(s', x)$ implies $\delta(s, x) \neq \delta(s', x)$. An input sequence w is *valid for* B, if for every prefix vx of w, where $v \in X^\star$ and $x \in X$, x is valid for the block $\delta(B, v)$.

2.2 Adaptive Distinguishing Sequence

Definition 1. *An* Adaptive Distinguishing Sequence *of an FSM* $M = (S, X, Y, \delta, \lambda)$ *with* n *states is a rooted tree* $\mathcal{A}$ *with* n *leaves such that: (i) Each leaf of* $\mathcal{A}$

is labeled by a distinct state $s \in S$. (ii) Each internal node of $\mathcal{A}$ is labeled by an input symbol $x \in X$. (iii) Each edge is labeled by an output symbol $y \in Y$. (iv) The outgoing edges of an internal node are labeled by distinct output symbols. (v) For a leaf node p labeled by a state $s \in S$, if w is the sequence of input symbols that appear in the nodes of the path from the root to p, and v is the sequence of output symbols labeling the edges of the path from root to p, then $\lambda(s, w) = v$.

An ADS $\mathcal{A}$ defines an adaptive experiment to identify the unknown initial state of an FSM, where the next input to be applied is decided by the input/output sequence observed previously. Let us assume that we are given an FSM M and we want to identify its current unknown state. One starts by applying the input symbol labeling the root of $\mathcal{A}$. We then follow the outgoing branch of the root that is labeled by y where y is the output symbol produced by M as the response to the input applied. This procedure is recursively applied for each subtree reached, until we reach a leaf node. The label of the leaf node gives the initial unknown state. An ADS of the FSM given in Figure 1 is given in Figure 2.

For a given deterministic FSM M, an ADS may or may not exist. One can check if M has an ADS in $O(mn \log n)$ time [23]. In this work, we consider only deterministic, minimal and completely specified FSMs, which are assumptions commonly adapted in the FSM based testing literature. Furthermore, we also consider only FSMs for which an ADS exist.

Let $\mathcal{A}$ be an ADS and p be a node in $\mathcal{A}$. We use *depth* of p to refer to the length of the path from the root of $\mathcal{A}$ to p. The *height* of $\mathcal{A}$ is defined to be the maximum depth of the leaves in $\mathcal{A}$. The *external path length* of $\mathcal{A}$ is the sum of depths of all the leaves in $\mathcal{A}$.

2.3 An Algorithm for Constructing an ADS

The LY algorithm [23] constructs an ADS for an FSM $M = (S, X, Y, \delta, \lambda)$ in two steps. First a tree called *Splitting Tree* (ST) is formed. For any block $B \subseteq S$ such that $|B| > 1$, ST provides a valid input sequence w for B such that $|\lambda(B, w)| > 1$. Hence, w is an input sequence that can partition B into smaller blocks as $B_1, B_2, \ldots, B_{|\lambda(B,w)|}$, where two states $s, s' \in B$ are in the same block B_i iff $\lambda(s, w) = \lambda(s', w)$.

In the second step of LY algorithm, ST is used to construct an ADS $\mathcal{A}$. The construction process explained below does not explicitly form an ADS as given in Definition 1, but it implies the construction of such a tree implicitly. The reader is referred to [23] for more details of the construction.

During the construction of the ADS, each nonleaf node p of $\mathcal{A}$ is labeled by an input sequence and each edge of $\mathcal{A}$ is labeled by an output sequence. Each node p of $\mathcal{A}$ is also associated with a block $\mathcal{C}(p)$ that corresponds to the current set of states. More precisely, if w is the concatenation of the input sequence labels of the nodes from the root to p (excluding the label of p), and v is the concatenation of the output labels of the edges from the root to p, then the block $\mathcal{C}(p)$ is set to $\mathcal{C}(p) = \{\delta(s, w) \mid v = \lambda(s, w), s \in S\}$.

Initially, the root node p of $\mathcal{A}$ is created by setting $\mathcal{C}(p) = S$, where S is the set of all states of M. As long as there is a leaf node p in the (partial) ADS tree constructed so far such that $|\mathcal{C}(p)| > 1$, the tree extended by processing such a node p. First an input sequence w is found from ST such that w is valid for $\mathcal{C}(p)$ and $|\lambda(\mathcal{C}(p), w)| > 1$. Then p is labeled by the input sequence w and a set of children node $p_1, p_2, \ldots, p_k$ of p is created, where a child node p_i is introduced for each possible output sequence $v \in \lambda(\mathcal{C}(p), w)$. The edge from p to p_i is labeled by v and the child p_i is associated with the block $\{\delta(s, w) \mid v = \lambda(s, w), s \in \mathcal{C}(p)\}$.

3 Proposed Algorithm

In this section, we explain the details of the proposed algorithm to construct a reduced ADS for an FSM $M = (S, X, Y, \delta, \lambda)$ which is minimal, deterministic, completely specified and known to have an ADS. Before presenting the details of the actual algorithm we provide some definitions and routines that are going to be used in this section.

For a block B, an input sequence w and an output sequence v, we use the notation $B_{w/v}$ to denote the set $B_{w/v} = \{s \in B \mid \lambda(s, w) = v\}$. In other words, $B_{w/v}$ is the set of states in B that produce the output sequence v when the input sequence w is applied.

In [28], Hennie introduces the use of a tree, called the *successor tree*, for constructing adaptive homing/distinguishing sequences. The successor tree grows exponentially and it possesses information that can be used to find the minimum cost adaptive homing/distinguishing sequences. However since it grows exponentially, it becomes impractical to construct a successor tree to obtain a reduced adaptive homing/distinguishing sequence for large FSMs.

Our method has two phases. In the first phase, a tree called the *Enhanced Successor Tree (EST)* similar to a successor tree is generated. In the second phase, EST is used to construct an ADS.

An EST contains two types of nodes; *input nodes* $\mathcal{I}$ and *output nodes* $\mathcal{O}$. The root and the leaves of an EST are output nodes. Except the leaves, the children of an output node are input nodes, and the children of an input node are output nodes. In other words, on a path from the root to a leaf, one observes a sequence of output and input nodes alternatingly, starting and ending with an output node.

Each input node p is labeled by an input symbol $in(p)$. Similarly, each output node q is labeled by an output symbol $out(q)$, except the root node for which $out(q) = \epsilon$. An output node q is also associated with a block $bl(q)$. For the root node q, $bl(q) = S$. An output node q is a leaf node iff $|bl(q)| = 1$. A non-leaf output node q that is associated with a block $bl(q)$, has a separate input node p as its child for each input symbol x that is valid for $bl(q)$, with $in(p) = x$. An input node p (where $x = in(p)$) with a parent output node q (where $B = bl(q)$), has a separate output node r as its child for each output symbol $y \in \lambda(B, x)$, with $out(r) = y$ and $bl(r) = \delta(B_{x/y}, x)$.

The EST of an FSM M is potentially an infinite tree. Instead of the whole tree, the algorithm constructs a limited size *partial* EST which can be used to construct an ADS. The algorithm uses heuristic approaches to explore the relevant and promising parts of the EST to find a reduced size ADS with respect to different metrics, such as the height and the external path length. The partial EST constructed by the algorithm will be the EST where the tree is pruned at several nodes. For a leaf node q in an EST we have $|bl(q)| = 1$. However, for a leaf node q in a partial EST, it is possible to have $|bl(q)| \geq 1$.

Initially, the algorithm starts with the partial EST consisting of only the root node. In each iteration, an output node q is handled and the partial EST rooted at q is expanded exhaustively upto depth k, where k is a parameter given to the algorithm. Among the children of q, an input node p that seems to be the best (according to the objective and the heuristic being used) is selected, and the search continues recursively under the subtrees rooted at the children of p.

During the construction of the partial EST T, some nodes are marked as the nodes to be used for ADS construction later. Namely, a set of output nodes L and a set of input nodes I are marked. Each output node $q \in L$ has the property that $|bl(q)| = 1$ and it corresponds to a leaf node in the ADS that will be constructed later. Also the nodes in I will be corresponding to the non–leaf nodes of the ADS. The algorithm constructing a partial EST is given in Algorithm 1.

Algorithm 1: Construct a partial EST for M

```
   Input: FSM M = (S, X, Y, δ, λ), k ∈ Z_{≥1}
   Output: A partial EST T, a set of leaves L, and a set of input nodes I
   begin
1      L ← ∅, I ← ∅
2      Construct an output node q_0 with bl(q_0) = S, out(q_0) = ϵ
3      Initialize T to be a tree consisting of the root q_0 only
4      Q ← {q_0} // Q is the set of output nodes yet to be processed
5      while Q ≠ ∅ do
6          Pick an output node q ∈ Q to process
7          Q ← Q \ {q}
8          ExpandEST(q,k) // expand subtree under q exhaustively upto a certain depth
9          Choose a child node p of q // based on the objective and the heuristic used
10         I ← I ∪ {p} // The input node p will be used for the ADS
11         foreach child r of p do
               if |bl(r)| > 1 then
12                 Q ← Q ∪ {r} // not a singleton yet, needs to be processed
               else
13                 L ← L ∪ {r} // reached a singleton block
```

The procedure "ExpandEST(q, k)", constructs the partial EST rooted at the node q exhaustively upto the given depth k. If in this partial subtree, for every leaf node r, we have $|bl(q)| = |bl(r)|$ (which means the block $bl(q)$ could not be divided into smaller blocks by using input sequences of length upto k that are valid for $bl(q)$), the procedure increases the depth of the subtree rooted at q until it encounters a level at which there exists a leaf node r with $|bl(r)| < |bl(q)|$. This is always possible since the FSM M has an ADS.

At line 9 of Algorithm 1, a child node p of q is chosen heuristically. This choice is based on the scores of the nodes which are calculated by processing the nodes in the subtree rooted at q in a bottom–up manner. First the scores of the

leaves in this subtree are computed. This is followed by the score evaluation of the internal nodes in the subtree. The score of a non–leaf node depends on the scores of its children. The score of a node reflects the potential size of the ADS that will be eventually formed if that node is decided to be used in the ADS to be formed.

When the score of an output node q is computed based on the scores of its children, since we have the control over the input to be chosen, the child node of q having the minimum score is chosen. However, when the score of an input node p is computed based on the scores of its children, since we do not have the control of the output to be produced, we prepare for the worst and use the maximum score of the children of p. A similar approach is in fact also suggested by Hennie (see Chapter 3 of [28]). The process of calculating the scores of the nodes depends on the heuristic used. The details are given in Section 3.1.

Before presenting the algorithm to construct an ADS, we will give some properties of the nodes L and I marked by Algorithm 1. For an output node q, consider the path from the root of T to q (including q). Let w and v be the concatenation of input symbols and output symbols on this path, respectively. We use below the notation $io(q)$ to refer to the input/output sequence w/v.

Proposition 1. *Let q be an output node in T and let $io(q) = w/v$. Then we have $bl(q) = \delta(S_{w/v}, w)$.*

Proof. The proof is trivial by using induction on the depth of q. □

Proposition 2. *$|L| + \sum_{q \in Q} |bl(q)| = |S|$ is an invariant of Algorithm 1 before and after every iteration the while loop.*

Proof. Before the first iteration, $|L| = 0$ and Q only has the root node q_0 for which we have $bl(q_0) = S$. In an iteration of the algorithm, an output node q is removed from Q and a child (input) node p of q is selected. It is sufficient to observe two facts. First, each state $s \in bl(q)$ is represented by a state $s' \in bl(q')$ where q' is a child of p, $s' = \delta(s, in(p))$ and $out(q') = \lambda(s, in(p))$. Second, no two states $s_1, s_2 \in bl(q)$ can be represented by the same state s' in the same child q', since $in(p)$ is a valid input for the states in $bl(q)$. Therefore, when we consider all children q' of p, we have $\sum_{q'} |bl(q')| = |bl(q)|$. Those children q' of p with $|bl(q')| = 1$ are included in L, and those children q' of p with $|bl(q')| > 1$ are included in Q. Hence the result follows. □

Proposition 2 implies the following result, since when Algorithm 1 terminates we have $Q = \emptyset$.

Corollary 1. *When Algorithm 1 terminates, $|L| = |S|$.*

Proposition 3. *Let q be an output node in T with $|bl(q)| = 1$, and let $w/v = io(q)$. There exists a unique state $s \in S$ such that $\lambda(s, w) = v$.*

Proof. Suppose s and s' are two distinct states such that $\lambda(s, w) = \lambda(s', w) = v$. By Proposition 1, we would then have $\delta(s, w)$ and $\delta(s', w)$ in $bl(q)$. Since $|bl(q)| = 1$, this implies $\delta(s, w) = \delta(s', w)$. This is not possible since at each step a valid input is applied, therefore no two states can be merged into a single state. □

Algorithm 2 describes how an ADS can be constructed based on the partial EST T, the set of marked nodes L and I in T by Algorithm 1. Note that at line 6 of Algorithm 2, $S_{w/v}$ is claimed to be a singleton, which is guaranteed by Proposition 3. In order to show that $\mathcal{A}$ which is generated by Algorithm 2 is an ADS, we also prove the following.

Proposition 4. *The leaves of $\mathcal{A}$ constructed by Algorithm 2 is labeled by distinct states.*

Proof. Assume that there are two leaf nodes q'_1 and q'_2 in $\mathcal{A}$ such that they are both labeled by the same state s. Let q_1 and q_2 be the leaf (output) nodes in T that correspond to q'_1 and q'_2. Let w_1/v_1 and w_2/v_2 be the input/output sequences $io(q_1)$ and $io(q_2)$. In this case, we would have $\lambda(s, w_1) = v_1$ and $\lambda(s, w_2) = v_2$. However, this is not possible since M is deterministic. □

Theorem 1. *$\mathcal{A}$ constructed by Algorithm 2 is an ADS.*

Proof. $\mathcal{A}$ has $n = |S|$ leaves as implied by Corollary 1. We will argue that $\mathcal{A}$ satisfies the conditions of Definition 1. Condition (i) is satisfied as shown by Proposition 4. Condition (ii) and Condition (iii) are easily satisfied due to the construction in Algorithm 2. Condition (iv) is satisfied due to the fact that in T, for an input node p, each child q of p has a distinct output symbol $out(q)$. Lines 5–8 of Algorithm 2, assign the label of leaves in such a way that Condition (v) is satisfied (see Proposition 3). □

3.1 Heuristics

We use different heuristic approaches to minimize the size of ADSs with respect to two different metrics, which are minimizing the height and minimizing the external path length of the ADS.

As mentioned above, the score of a node q in the partial EST constructed so far is an estimation of the size of the ADS that will be formed by using a child of q in ADS. Let $d(q)$ be the depth of q in the partial EST and $v(q)$ be the score of q. We also keep track of another information, $z(q)$. It is the number of singleton output nodes in the winner subtrees under q in the current partial EST, and used to break the ties as explained below.

Let us consider a leaf node q, where $|bl(q)| = 1$. This is in fact a leaf node also in the complete EST. For such leaf nodes, we set $v(q) = d(q)$ and $z(q) = 1$. However, for a leaf node q in the current partial EST with $|bl(q)| > 1$, we set $z(q) = 0$. Although q is currently a leaf node in the partial EST, if we were to expand q, there will appear a subtree under q. In order to take into account the size of the subtree rooted at q (without actually constructing this subtree), we need to estimate the size of the subtree under q. Note that $bl(q)$ is the set of states yet to be distinguished from each other.

We consider two different metrics as the size of an ADS: height or external path length. Depending on the objective, we estimate the size of the subtree

Algorithm 2: Construct an ADS

Input: The partial EST T, the set of marked nodes L and I by Algorithm 1
Output: An ADS $\mathcal{A}$
begin
    `// Construct and label the internal nodes of` $\mathcal{A}$
1   **foreach** node $p \in I$ **do**
2   Construct an internal node p' in $\mathcal{A}$
3   Label p' by $in(p)$
    `// Construct and label the leaf nodes of` $\mathcal{A}$
4   **foreach** node $q \in L$ **do**
5   Let $w/v = io(q)$
6   Let s be the state such that $\{s\} = S_{w/v}$
7   Construct a leaf node q' in $\mathcal{A}$
8   Label q' by s
    `// Construct the edges to the leaves`
9   **foreach** leaf node $q' \in \mathcal{A}$ **do**
10   Let q be the corresponding node of q' in T
11   Let p be the parent of q in T
12   Let p' be the corresponding node of p in $\mathcal{A}$
13   Insert an edge between p' and q' with the label $out(q)$
    `// Construct the remaining edges`
14 **foreach** internal node $p' \in \mathcal{A}$ **do**
15   Let p be the corresponding node of p' in T
16   **if** *p has a grandparent in T* **then**
            `// except the root of` $\mathcal{A}$
17   Let q be the parent of p in T
18   Let r be the parent of q in T
19   Let r' be the corresponding node of p in $\mathcal{A}$
20   Insert an edge between p' and r' with the label $out(q)$

that would appear under a (yet to be processed) output node q in different ways. While minimizing for height, we use two different heuristic functions $\mathcal{H}_U : \mathcal{O} \to \mathbb{R}^+$ and $\mathcal{H}_{LY} : \mathcal{O} \to \mathbb{R}^+$. Similarly, while optimizing for external path length, we use heuristic functions $\mathcal{L}_U : \mathcal{O} \to \mathbb{R}^+$ and $\mathcal{L}_{LY} : \mathcal{O} \to \mathbb{R}^+$.

For an FSM with n states the height of an ADS is bounded above by $n(n-1)/2$ [23]. We use this bound for heuristic functions $\mathcal{H}_U$ and $\mathcal{L}_U$ in the following way. The score of node q with respect to function $\mathcal{H}_U$ is given as

$$\mathcal{H}_U(q) = d(q) + |bl(q)|(|bl(q)| - 1)/2$$

where $d(q)$ is the depth of q. On the other hand, function $\mathcal{L}_U$ multiplies the number of states with the expected height of the subtree to approximate the expected external path length. That is

$$\mathcal{L}_U(q) = (d(q) + |bl(q)|(|bl(q)| - 1)/2)|bl(q)|$$

As another estimation method for the size of the subtree to appear under an output node q, one can use LY algorithm to construct an ADS for the states in $bl(q)$. The heuristic functions $\mathcal{H}_{LY}$ and $\mathcal{L}_{LY}$ use this idea. Let A' be the ADS computed by LY algorithm for the states in $bl(q)$, and let $h(A')$ and $l(A')$ be the height and the external path length of A'. Then the heuristic functions $\mathcal{H}_{LY}$

and $\mathcal{L}_{LY}$ are defined as

$$\mathcal{H}_{LY}(q) = d(q) + h(A')$$
$$\mathcal{L}_{LY}(q) = d(q)|bl(q)| + l(A')$$

At line 9 of Algorithm 1, for the output node q being processed in that iteration, an input node p which is a child of q is chosen. Let T' refer to the subtree rooted at q in the partial EST at this point. While choosing the child input node p to be used, the scores of the nodes in T' are calculated in a bottom up manner. First, for each (current) leaf node q' (which is an output node) in T', $v(q')$ is assigned by using one of the heuristic functions ($\mathcal{H}_U(q')$ or $\mathcal{H}_{LY}(q')$ for height optimization, and $\mathcal{L}_U(q')$ or $\mathcal{L}_{LY}(q')$ for external path length optimization) and $z(q')$ is assigned. The score of the remaining nodes in T' are based on the scores of its children and are calculated as follows.

When minimizing for height, for an input node p', $v(p')$ is set to the maximum score of its children and $z(p')$ is set to the sum of singleton scores of its children. For an output node q', $v(q')$ is set to the minimum score of its children, and $z(q')$ is set to the $z(.)$ value of the winner child. When minimizing for external path length, for an input node p', $v(p')$ and $z(p')$ is set to the sum of the scores of its children. For an output node q' on the other hand, $v(q')$ is set to the minimum score of its children and $z(q')$ is set to the $z(.)$ value of the winner child. Note that, there may be ties during this process when we attempt to take minimum or maximum. Among the nodes achieving the same minimum/maximum, the tie is first tried to be broken by maximizing the number of singleton values ($z(.)$). If there is still a tie, this is broken randomly.

4 Experiments

We generated two test suites, TS1 and TS2. In each test suite, we have 6 classes of FSMs. Each class contains 100 FSMs. Thus the number of FSMs used in these experiments is 600(TS1)+600(TS2) = 1200. In TS1, the state and input/output alphabet cardinalities of the classes are $(30, 4/4)$, $(30, 8/8)$, $(30, 16/16)$, $(60, 4/4)$, $(60, 8/8)$ and $(60, 16/16)$. TS1 is used to see the effect of the changes in the input/output alphabet cardinalities. For these experiments, the value of the lookahead parameter k is set to 1.

In order to see the effect of the changes in the number of states and the value of the lookahead parameter k, we use the FSMs in TS2. In TS2, the input/output alphabet cardinalities are fixed to $(2/2)$ and the state cardinalities of the classes are $\{25, 30, \ldots, 50\}$. For these experiments, the value of k varies between 1 and 3. Note that while computing an ADS for an FSM M, our heuristic reduces to the brute–force approach when k is greater than or equal to the height of the minimum ADS of M. Hence for such FSMs, our heuristic actually gives exact results. In order to see the effect of the parameter k, we need to have FSMs such that the height of the minimum ADS is larger than the maximum k value used in the experiments. Note that, even for the smallest number of states used in TS2 ($n = 25$), the height of the minimum ADS cannot be smaller than $\log_2 25 > 4$.

4.1 FSM Generation

We randomly generated FSMs using the tool utilised in [15, 29]. To construct an FSM M, first, for each input x and state s we randomly assign the values of $\delta(s, x)$ and $\lambda(s, x)$. If M is strongly connected[2], is minimal, and has an ADS[3], M is included into the test suite. Otherwise, it is discarded. We use Intel Xeon E5-1650 @3.2-GHZ CPU with 16 GB RAM to carry out these tests. We implemented proposed algorithm, LY algorithm and the brute-force algorithm using C++ and compiled them using Microsoft Visual Studio .Net 2012 under 64 bit Windows 7 operating system.

4.2 Evaluation

In order to evaluate the relative performance of different approaches, we compute the ADSs using the proposed algorithm with heuristic functions $\mathcal{H}_U, \mathcal{L}_U, \mathcal{H}_{LY}$ and $\mathcal{L}_{LY}$. Also for each FSM we compute an ADS by using LY and brute–force algorithms. The brute–force algorithm (BF) is described in [26]. BF algorithm constructs EST to a depth that is sufficient to form an ADS. Therefore it is an exponential time algorithm.

In the following sections we present the results of our experimental study. We explain below the measures that we use to compare the relative performance of the algorithms. Here, A refers to the heuristics used (i.e. A is one of $\mathcal{H}_U, \mathcal{L}_U, \mathcal{H}_{LY}$ and $\mathcal{L}_{LY}$). We use $size$ to refer to the height or the external path length, depending on the objective of the minimization. For example, for a given FSM M, if we are considering heights, $size(\mathcal{H}_{LY}(M))$ is the height of the ADS computed by $\mathcal{H}_{LY}$, $size(LY(M))$ is the height of the ADS computed by LY algorithm, and $size(BF(M))$ is the height of the ADS computed by BF algorithm. In this case, $\mathcal{G}_1(\mathcal{H}_{LY}, M)$ gives the percentage decrease in the height of the ADS computed by $\mathcal{H}_{LY}$ compared to the height of the ADS computed by LY algorithm. Therefore, higher $\mathcal{G}_1$ values indicate a better performance of the heuristics over LY algorithm.

Similarly, $\mathcal{G}_2(\mathcal{H}_{LY}, M)$ gives the percentage decrease in the height of the ADS constructed by BF algorithm compared to the height of the ADS constructed by $\mathcal{H}_{LY}$. Therefore, lower $\mathcal{G}_2$ values indicate a better performance of the heuristics, approaching to the optimal values computed by BF algorithm.

Finally, $time(LY(M))$ and $time(A(M))$ are used to denote the time required to compute an ADS by LY and a heuristics, respectively. Although, we compare the height and external path length performance with respect to BF, we only compare the time performance of our heuristics with respect to LY algorithm. This is because, for large FSMs, the exponential BF algorithm will take too much time and will be impractical.

---

[2] M is strongly connected if for any pair (s, s') of states of M, there exists an input sequence w such that $\delta(s, w) = s'$.

[3] Existence check for an ADS can be performed in $O(mn \log n)$ time [23].

$$\mathcal{G}_1(A,M) = \frac{size(LY(M)) - size(A(M))}{size(LY(M))} \times 100$$
$$\mathcal{G}_2(A,M) = \frac{size(A(M)) - size(BF(M))}{size(A(M))} \times 100$$
$$\mathcal{T}(A,M) = \frac{time(LY(M)) - time(A(M))}{time(LY(M))} \times 100$$

A positive $\mathcal{T}$ value for a heuristic means that the heuristic computes an ADS faster than LY algorithm. Hence higher $\mathcal{T}$ values are better.

4.3 Results

The effect of the value of the lookahead parameter k We discuss the effect of the lookahead distance, i.e. the value of the parameter k used in Algorithm 1, by using the experimental results given in Table 1. As expected, higher values of k yield a better performance since the heuristics look further into the EST tree to be generated. The results of the measure $\mathcal{G}_1$ is directly proportional, and the results of the measure $\mathcal{G}_2$ are inversely proportional to k. That is, as the value of parameter k increases, regardless to the heuristic function used, the algorithm produces cheaper ADSs than the LY algorithm. Moreover the results suggest that the difference between the ADS costs constructed by the proposed heuristics and the BF algorithm reduces as k increases. However, the time needed to compute ADSs increases with the values of k. Therefore these experiments reveal that increasing the value of k, improves the quality of the results at the expense of increased running times.

The effect of the number of states Again using the results given in Table 1, we see that the results of the measure $\mathcal{G}_1$ increase with the number of states, especially for higher values of k regardless to the heuristic function used. Therefore we deduce that the performance of all heuristic functions (compared to the performance of LY algorithm) increases as the number of states increases. Note that as the number of states increases, the height and the external path length of optimal ADSs also increase. For such ADSs, our lookahead algorithms start to perform even better compared to LY algorithm, since for such ADSs the information provided by the lookahead algorithms becomes more effective in guiding the search.

On the other hand, the results of the measure $\mathcal{G}_2$ also increase with the number of states. This implies that as the number of states increases the costs of the ADSs computed by proposed heuristics diverge from the costs of optimal ADSs computed by the BF. This is also expected, since for a constant k value, as the height of minimum ADSs increases (which is the case when the number of states increases), the effectiveness of looking k steps ahead reduces.

In terms of measure $\mathcal{T}$, we surprisingly see that, heuristics using approximation (i.e. $\mathcal{H}_U$ and $\mathcal{L}_U$) are faster than LY algorithm. Importantly, the values of measure $\mathcal{T}$ are directly proportional to the number of states for $\mathcal{H}_U$ and $\mathcal{L}_U$.

Note that, FSMs in TS2 has two input symbols only. Therefore, $\mathcal{H}_U$ and $\mathcal{L}_U$ have to consider only two different input symbols. As the number of input symbols increase, $\mathcal{H}_U$ and $\mathcal{L}_U$ will start to get slower, as also supported by our discussion given below for the effect of the number of input/output symbols on the performance of the heuristics.

The heuristic functions that use LY algorithm (i.e. $\mathcal{H}_{LY}$ and $\mathcal{L}_{LY}$) are the slowest approaches and the values of the measure $\mathcal{T}$ are indirectly proportional to the number of states. This is quite expected, since these heuristics already use LY algorithm many times during their execution.

The effect of the size of the input/output alphabet In order to see the effect of the size of the input/output alphabet, we performed experiments using FSMs in TS1 and by setting $k = 1$. The results of the experiments are given in Table 2. We observe that as the input/output alphabet cardinalities increase, the performance of the heuristics improve with respect to both $\mathcal{G}_1$ and $\mathcal{G}_2$ measures. The reason for this performance increase is that, when there are more input symbols, the search space of our heuristics also increases, which allow heuristics to pick a better option. However, this comes with the cost of increased running time, as the $\mathcal{T}$ measure rows in Table 2 display.

The effect of the heuristic used $\mathcal{L}_U$ and $\mathcal{H}_U$ are better in their time performances over $\mathcal{L}_{LY}$ and $\mathcal{H}_{LY}$. For both height and external path length minimization, the heuristics based on approximation (i.e. $\mathcal{H}_U$ and $\mathcal{L}_U$) show a similar performance to the other heuristics based on LY algorithm (i.e. $\mathcal{H}_{LY}$ and $\mathcal{L}_{LY}$), regardless of the number of states and the value of k.

Based on these results, the heuristics $\mathcal{H}_U$ and $\mathcal{L}_U$ can be preferred over the heuristics $\mathcal{H}_{LY}$ and $\mathcal{L}_{LY}$.

5 Discussions and Conclusions

In this paper we propose a lookahead-based algorithm for constructing reduced cost ADSs. Since an ADS is a tree, we consider the height and the sum of all paths from the root to the leaves (external path length) as the cost of an ADS. The proposed method uses a tree (called EST) to lookahead while constructing an ADS. The construction of the tree is flexible and allows a trade of between the cost of the ADS and the time required to construct the ADS by changing the value of the lookahead parameter. Based on the objective of the minimization (height or external path length), the proposed algorithm uses different heuristic functions.

We perform experimental study to evaluate the different heuristics used by the proposed algorithm and compare it to the optimal solutions. The results indicate that the proposed algorithm tends to construct optimum ADSs as we increase the value of the lookahead parameter. This is natural, since as the lookahead parameter value increases, our algorithm approaches to the brute

$Measure$	k	n					
		25	30	35	40	45	50
$\mathcal{G}_1(\mathcal{L}_U, M)$	1	5.654	3.420	5.879	4.987	6.925	5.244
	2	6.417	6.353	6.102	6.417	7.058	9.131
	3	8.027	8.025	8.009	8.457	7.713	10.650
$\mathcal{G}_1(\mathcal{L}_{LY}, M)$	1	6.994	7.105	7.288	7.310	6.896	8.019
	2	7.745	7.812	7.806	8.039	7.497	10.712
	3	7.848	8.240	8.235	8.515	7.880	10.243
$\mathcal{G}_1(\mathcal{H}_U, M)$	1	10.503	14.092	13.069	10.327	11.268	9.730
	2	11.883	11.434	9.9852	12.227	10.384	14.554
	3	13.339	15.580	16.412	16.022	15.467	18.889
$\mathcal{G}_1(\mathcal{H}_{LY}, M)$	1	11.162	14.541	14.951	14.934	14.410	17.751
	2	13.676	16.246	17.764	17.283	16.457	20.605
	3	13.968	16.955	18.760	17.874	17.341	21.439
$\mathcal{G}_2(\mathcal{L}_U, M)$	1	3.338	3.874	2.716	3.883	1.472	6.794
	2	1.883	2.259	2.577	2.587	1.331	2.639
	3	0.249	0.520	0.537	0.433	0.585	0.971
$\mathcal{G}_2(\mathcal{L}_{LY}, M)$	1	1.352	1.491	1.320	1.668	1.464	3.68
	2	0.560	0.742	0.764	0.884	0.822	0.937
	3	0.446	0.282	0.300	0.368	0.413	0.355
$\mathcal{G}_2(\mathcal{H}_U, M)$	1	3.545	3.783	6.823	8.764	6.911	8.975
	2	2.797	1.053	4.766	6.794	5.634	8.538
	3	1.587	1.746	2.841	3.102	2.238	3.672
$\mathcal{G}_2(\mathcal{H}_{LY}, M)$	1	3.262	3.020	4.405	4.241	3.441	5.138
	2	0.511	1.014	1.408	1.607	1.238	1.790
	3	0.219	0.176	0.297	0.959	0.190	0.817
$\mathcal{T}(\mathcal{L}_U, M)$	1	55.213	41.567	50.706	55.688	63.942	63.141
	2	43.112	35.449	26.101	47.613	49.851	55.000
	3	27.234	21.862	12.617	21.916	36.237	37.500
$\mathcal{T}(\mathcal{L}_{LY}, M)$	1	-51.1112	-80.335	-105.958	-99.978	-105.833	-103.653
	2	-107.422	-128.748	-192.155	-158.952	-161.000	-175.598
	3	-140.556	-208.307	-234.701	-271.012	-255.302	-264.040
$\mathcal{T}(\mathcal{H}_U, M)$	1	57.112	54.123	58.620	50.334	61.622	64.624
	2	42.887	40.777	37.077	45.467	54.385	53.400
	3	21.775	36.451	12.093	32.654	29.290	28.232
$\mathcal{T}(\mathcal{H}_{LY}, M)$	1	-65.332	-91.334	-122.624	-99.458	-103.152	-120.690
	2	-102.962	-148.223	-166.762	-175.744	-165.978	-188.476
	3	-156.223	-212.169	-232.074	-269.048	-239.374	-268.426

Table 1: Comparison of ADS costs and computation times of algorithms with respect to differing state cardinalities and lookahead paramater values (k).

$Measure$	i/o	n	
		30	60
$\mathcal{G}_1(\mathcal{L}_U, M)$	4/4	6.821	6.367
	8/8	8.300	9.967
	16/16	12.408	11.174
$\mathcal{G}_1(\mathcal{L}_{LY}, M)$	4/4	5.839	7.920
	8/8	8.811	9.536
	16/16	12.467	10.842
$\mathcal{G}_1(\mathcal{H}_U, M)$	4/4	10.150	9.561
	8/8	16.466	21.250
	16/16	23.166	34.666
$\mathcal{G}_1(\mathcal{H}_{LY}, M)$	4/4	17.283	19.895
	8/8	17.300	21.500
	16/16	23.939	35.166
$\mathcal{G}_2(\mathcal{L}_U, M)$	4/4	2.204	3.334
	8/8	0.457	0.117
	16/16	0.018	0.041
$\mathcal{G}_2(\mathcal{L}_{LY}, M)$	4/4	1.666	1.722
	8/8	1.054	0.491
	16/16	0.413	0.508
$\mathcal{G}_2(\mathcal{H}_U, M)$	4/4	2.666	1.600
	8/8	1.110	1.333
	16/16	0.333	0.250
$\mathcal{G}_2(\mathcal{H}_{LY}, M)$	4/4	2.750	1.200
	8/8	2.000	0.000
	16/16	0.000	0.666
$\mathcal{T}(\mathcal{L}_U, M)$	4/4	45.234	11.000
	8/8	-12.223	-11.223
	16/16	-24.122	-38.330
$\mathcal{T}(\mathcal{L}_{LY}, M)$	4/4	-34.563	-226.500
	8/8	-145.221	-300.199
	16/16	-167.331	-391.442
$\mathcal{T}(\mathcal{H}_U, M)$	4/4	32.568	10.000
	8/8	-55.332	-131.120
	16/16	-77.127	-167.883
$\mathcal{T}(\mathcal{H}_{LY}, M)$	4/4	-45.223	-204.500
	8/8	-184.331	-315.123
	16/16	-199.221	-412.774

Table 2: Comparison of ADS costs and computation times of algorithms with respect to differing input output alphabet cardinalities.

force algorithm, hence the time requirement of our algorithm also increases. However, even with small lookahead parameter values, we efficiently construct ADSs smaller than those constructed by LY algorithm. In some cases, we observe that we can construct smaller ADSs faster than LY algorithm.

As a future work, we are planing to extend the experiments for larger FSMs. Moreover an evaluation is required to see the effect of using reduced ADSs for constructing test sequences. Finally, it would be interesting to extend this work to non-deterministic FSMs.

Acknowledgments

This work is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant #113E292.

References

1. A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. In *Protocol Specification, Testing, and Verification VIII*, pages 75–86, Atlantic City, 1988. Elsevier (North-Holland).
2. R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 98–109. ACM, 2005.
3. A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 248–257. IEEE Computer Society, 2004.
4. T. S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.
5. I. Pomeranz and S. M. Reddy. Test generation for multiple state-table faults in finite-state machines. *IEEE Transactions on Computers*, 46(7):783–794, 1997.
6. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27:218–228, July 2002.
7. K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks*, 15(4):285–297, 1988.
8. F. C. Hennie. Fault-detecting experiments for sequential circuits. In *Proceedings of Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, Princeton, New Jersey, November 1964.
9. G. Gonenc. A method for the design of fault detection experiments. *IEEE Transactions on Computers*, 19:551–558, 1970.
10. M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics and Systems Analysis*, 9:653–665, 1973. 10.1007/BF01068590.
11. S. T. Vuong, W. W. L. Chan, and M. R. Ito. The UIOv-method for protocol test sequence generation. In *The 2nd International Workshop on Protocol Test Systems*, Berlin, 1989.
12. S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

13. H. Ural and K. Zhu. Optimal length test sequence generation using distinguishing sequences. *IEEE/ACM Transactions on Networking*, 1(3):358–371, 1993.
14. A. Petrenko and N. Yevtushenko. Testing from partial deterministic FSM specifications. *IEEE Transactions on Computers*, 54(9):1154–1165, 2005.
15. R. M. Hierons, G. V. Jourdan, H. Ural, and H. Yenigun. Checking sequence construction using adaptive and preset distinguishing sequences. In *Proceedings of 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009)*, pages 157–166. IEEE Computer Society, 2009.
16. R. M. Hierons and H. Ural. Optimizing the length of checking sequences. *IEEE Transactions on Computers*, 55:618–629, May 2006.
17. R. M. Hierons and H. Ural. Reduced length checking sequences. *IEEE Transactions on Computers*, 51(9):1111–1117, 2002.
18. R. M. Hierons. Minimizing the number of resets when testing from a finite state machine. *Information Processing Letters*, 90(6):287–292, 2004.
19. H. Ural, X. Wu, and F. Zhang. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93–99, 1997.
20. Z. Kohavi. *Switching and Finite State Automata Theory*. McGraw-Hill, New York, 1978.
21. R. T. Boute. Distinguishing sets for optimal state identification in checking experiments. *IEEE Trans. Comput.*, 23:874–877, 1974.
22. G. V. Jourdan, H. Ural, H. Yenigun, and J. Zhang. Lower bounds on lengths of checking sequences. *Formal Aspects of Computing*, 22(6):667–679, 2010.
23. D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.
24. M. N. Sokolovskii. Diagnostic experiments with automata. *Cybernetics and Systems Analysis*, 7:988–994, 1971.
25. N. Kushik, K. El-Fakih, and N. Yevtushenko. Adaptive homing and distinguishing experiments for nondeterministic finite state machines. In H. Yenigun, C. Yilmaz, and A. Ulrich, editors, *Testing Software and Systems*, volume 8254 of *Lecture Notes in Computer Science*, pages 33–48. Springer Berlin Heidelberg, 2013.
26. A. Gill. *Introduction to The Theory of Finite State Machines*. McGraw-Hill, New York, 1962.
27. U. C. Türker and H. Yenigun. Hardness and inapproximability of minimizing adaptive distinguishing sequences. *Formal Methods in System Design*, 44(3):264–294, 2014.
28. F.C. Hennie. *Finite-state models for logical machines*. Wiley, 1968.
29. C. Gunicen, U. C. Türker, H. Ural, and H. Yenigün. Generating preset distinguishing sequences using SAT. In E. Gelenbe, R. Lent, and G. Sakellari, editors, *Computer and Information Sciences II*, pages 487–493. Springer London, 2012. 10.1007/978-1-4471-2155-8_62.